

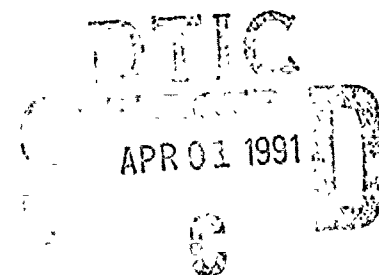
Final Report
for the period
March 1990 to
October 1990

GPS - A PostScript-like Language for System Simulation

January 1991

Author:
H.K. Geyer

Argonne National Laboratory
9700 South Cass Avenue
Argonne IL 60437-4841



Approved for Public Release

Distribution is unlimited. The OL-AC/PL Technical Services Office has reviewed this report and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

Prepared for the: **OL-AC, Phillips Laboratory (AFSC)**
Air Force Systems Command
Edwards AFB CA 93523-5000

91 3 28 092

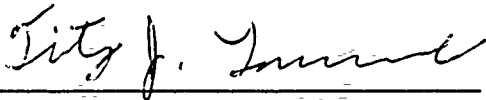
NOTICE

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related Government procurement operation, the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise, or in any way licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use or sell any patented invention that may be related thereto.

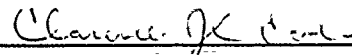
FOREWORD

This final report was submitted by Argonne National Laboratory, Argonne IL on completion of Project AFAL-89016 with the OL-AC, Phillips Laboratory (AFSC) (formerly Astronautics Laboratory), Edwards AFB CA 93523-5000. Argonne National Laboratory is operated by the University of Chicago for the United States Department of Energy under Contract W-31-109-Eng-38. OLAC PL Project Manager was Lt Tim Lawrence.

This report has been reviewed and is approved for release and distribution in accordance with the distribution statement on the cover and on the DD Form 1473.



TIMOTHY J. LAWRENCE, LT, USAF
Project Manager



CLARENCE J.C. COLEMAN, Capt, USAF
Chief, Advanced Concepts Branch



DAVID W. LEWIS, Maj, USAF
Director of Advanced Programs

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S) AL-TR-90-085		
4. PERFORMING ORGANIZATION REPORT NUMBER(S)			7a. NAME OF MONITORING ORGANIZATION Phillips Laboratory		
6a. NAME OF PERFORMING ORGANIZATION Argonne National Laboratory		6b. OFFICE SYMBOL (If applicable) ANL	7b. ADDRESS (City, State, and ZIP Code) OL-AC PL/LSVF Edwards AFB, CA 93523-5000		
6c. ADDRESS (City, State, and ZIP Code) 9700 South Cass Avenue Argonne, IL 60437-4841		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER AFAL 89016			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)	10. SOURCE OF FUNDING NUMBERS		
8c. ADDRESS (City, State, and ZIP Code)		PROGRAM ELEMENT NO. 62302F	PROJECT NO. 3058	TASK NO. 00	WORK UNIT ACCESSION NO. 7D
11. TITLE (Include Security Classification) GPS - A Postscript - like Language for System Simulations					
12. PERSONAL AUTHOR(S) Howard K. Geyer					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM 90/03 TO 90/10		14. DATE OF REPORT (Year, Month, Day) 91/01	
15. PAGE COUNT 108		16. SUPPLEMENTARY NOTATION OL-AC PL was formerly the Astronautics Laboratory (AFSC)			
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Mission analysis, nuclear propulsion, thermalhydraulics		
21	06				
19. ABSTRACT (Continue on reverse if necessary and identify by block number) GPS is a post script (a registered trademark of Adobe Systems, Inc.) like language which can be used to interact with the models and mathematical utility classes used with the SALT (System Analysis Language Translator) code. This permits writing drivers that are interpreted rather than compiled, as with the current version of SALT, thus saving the resulting compile and load times on the computer. Another advantage that GPS affords over the SALT code is its ability to interrupt the execution of a system problem and then query and change system variables. In order to use GPS with a set of models, there are several requirements to which the models must conform. However these requirements are not complex, and in general, simply amount to adding a mechanism to locate model variables and functions via their names. GPS was developed as an alternative to directly writing drivers for the SALT code (C++ version) and is itself one of several alternative ways of developing direct methods of doing system studies (as opposed to indirect methods in which the specially developed driver for the system must first be compiled). For example, a general purpose					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL Lt Timothy J. Lawrence			22b. TELEPHONE (Include Area Code) 805-275-5640		22c. OFFICE SYMBOL LSVF

19. ABSTRACT Cont.

driver could be written and linked to the SALT models which would provide a number of different system configurations, possible system constraints, parametric studies, etc. This is an approach often employed by system codes and results in the easiest to use code in as much as the inputs must take a fixed format. This approach also makes for a robust code. However, even the most general hard wired drivers have limitations which are usually encountered eventually. Another alternative would be to develop a general purpose interpreted language for interacting with system models. This provides for handling a greater range of potential system problems provided that the developed language is sufficiently flexible. In order to obtain this flexibility most of the capabilities of a general purpose programming language must be provided, this means that a relatively sophisticated language parser must be developed. As a third alternative, a language like post script or Forth might be employed in which there is relatively little syntax structure and thus does not need a sophisticated parser. These languages are also extensible providing even greater flexibility, however; these languages have a disadvantage in that they are often hard to read due to their use of postfix notation and of stacks. While these disadvantages cannot be entirely eliminated, with some suitable extensions to the language and some discipline in writing input they can be reduced. GPS makes use of this last alternative. The first section presents an overview of the GPS language and some of its differences with postscript. The second system presents a brief discussion of the Postscript like language operators followed by a section on the new operators for dealing with models developed.



Approved for Release	
SECRET	<input checked="" type="checkbox"/>
CONFIDENTIAL	<input type="checkbox"/>
NOFORN	<input type="checkbox"/>
Date: 10/10/00	
By: [Signature]	
Remarks: [Handwritten]	
A-1	

Table of Contents

Chapter 1 GPS Language	1
1.1 Introduction	1
1.2 GPS Overview	2
1.3 PostScript like operators	2
1.4 Additional GPS Operators	7
Chapter 2 General Model and Mathematical Utilities Classes	11
2.1 Introduction	11
2.2 Class conventions	11
2.3 C Stacks	11
2.4 Task class	12
2.5 Task Class Examples	15
2.5.1 Example one	15
2.5.2 Example two	15
2.5.3 Example three	16
2.5.4 Example four	17
2.5.5 Example five	18
2.5.6 Example six	19
Chapter 3 Steady-State Power System Model Classes	21
3.1 Introduction	21
3.2 Steady-State Model Flow classes	21
3.3 Steady-state models	23
3.3.1 Gas (gas) model	23
3.3.2 Mixer (mx) model	24
3.3.3 Splitter (sp) model	25
3.3.4 Heater (ht) model	26
3.3.5 Gas turbine (gt) model	27
3.3.6 Gas compressor (cp) model	27
3.3.7 Heat exchanger (hx) model	28
3.3.8 Pump (pump) model	29
3.3.9 Diffuser (df) model	29
3.3.10 Nozzle (nz) model	30
3.3.11 Combustor (cb) model	30
3.3.12 Power (power) model	32
3.4 Steady-State Example One	32
3.5 Steady-State Example Two	34
3.6 Steady-State Example Three	34
3.7 Steady-State Example Four	35

Chapter 4 Dynamic Power System Model Classes	39
4.1 Introduction	39
4.2 Dynamic Model Flow Classes	39
4.3 Dynamic Models	40
4.3.1 Gas (gas) model	40
4.3.2 Shaft (shft) model	41
4.3.3 Compressor (cp) model	42
4.3.4 Exhaust nozzle (exnz) model	43
4.3.5 Gas turbine (gt) model	45
4.3.6 Heater (ht) model	46
4.3.7 Mixer (mx) model	48
4.3.8 Pipe (pi) model	48
4.3.9 Pump (pump) model	49
4.3.10 Reactor (reac) model	50
4.3.11 Splitter (sp) model	52
4.3.12 Valve (valv) model	53
4.3.13 Motor (mot) model	53
4.3.14 Generator (gen) model	54
4.3.15 Controller (cntl) model	54
4.4 Dynamic system tasks	55
4.5 Dynamic example one	56
Chapter 5 Thermionic Power System Model Classes	61
5.1 Introduction	61
5.2 Thermionic Model Flow Classes	61
5.3 Thermionic models	61
5.3.1 Reactor (reac) model	62
5.3.2 Thermionic Converter (ti) model	63
5.3.3 Radiator (rad) model	63
5.3.4 Boost Converter (bc) model	64
5.3.5 Flow Splitter (sp) model	64
5.3.6 Resistor (res) model	65
5.3.7 Bus (bus) model	65
5.3.8 Mass (mass) model	65
5.4 Thermionic System Example	65
Chapter 6 Graphics	67
6.1 Introduction	67
6.2 Two-dimensional Plots	67
6.3 System Diagrams	68
6.3.1 Configuration Windows	68
6.3.2 Diagram Editing	69
Chapter 7 GPS Model Interfacing	71
7.1 Introduction	71
7.2 Interfacing example	71

7.3 Other requirements	73
References	75
Appendix A Task Class examples	76
Appendix B Steady-State Example Four	85
Appendix C Steady-State Example Four with Constraints	86
Appendix D Dynamic Example One at Design Point	88
Appendix E Dynamic Example One	91
Appendix F Thermionic System Example	97
Appendix G Performance Map Layouts	99

CHAPTER 1

GPS Language

1.1. Introduction

GPS is a PostScript¹ like language for performing steady-state and dynamic system simulations. The systems that can be analyzed consist of arbitrary configurations of component models interconnected by various flows. The component models can also be quite arbitrary in their modeling sophistication with new models being easily added by the user. The flows are likewise arbitrary in their complexity and may represent actual physical flows, such as gasses or liquids in the case of a power system, or simply the flow of information in the case of a sensor or control system. GPS also permits the user to include arbitrary system constraints over part or all of the system under consideration. In fact, different system constraints may be imposed on any collection of user defined nested or unnested subsystems. Additionally, GPS permits the user to define nonlinear objective functions to be minimized subject to both equality or inequality nonlinear constraints over any collection of subsystems. These types of capabilities permit one to quickly set up and perform an almost endless variety of system scenario studies.

GPS was developed to make use of the various component models previously developed for use within the SALT (System Analysis Language Translator) code [1,2]. Thus, a number of component models for power systems are currently available. In addition, a number of fluid properties procedures are also available for use with those models. Several of these model libraries are described in this report. In order to use GPS with any set of models, there are several requirements to which the models must conform. However, these requirements are not complex, and, in general, simply amount to adding a mechanism to locate model variables and functions via their names.

GPS has a number of advantages over the SALT code. SALT originally made use of a preprocessor technique in its PL/I version and in its C++ version required the user to simply write a driver using the various C++ model classes. In either case the resulting driver had to be compiled and then linked to the component models, mathematical utilities, property procedures, etc. GPS drivers are interpreted rather than compiled, thus saving the resulting compile and load times on the computer, and permitting very rapid turn-around times in performing a series of system studies. Another advantage that GPS affords over the SALT code is its ability to interrupt the execution of a system problem and then query and change system variables.

As an alternative to directly writing drivers for the SALT code, GPS is itself one of several alternative ways of developing non-preprocessor methods for analyzing systems. For example, a general purpose driver could be written and linked to the models which would provide a number of different system configurations, possible system constraints, parametric studies, etc. This is an approach often employed by system codes and results in the easiest to use code in as much as the inputs must take a fixed format. This approach also makes for a robust code. However, even the most general hardwired drivers have limitations which are usually encountered eventually. Another alternative would be to develop a general purpose interpreted language for interacting with system models. This provides for handling a greater range of potential system problems provided that the developed language is sufficiently flexible. In order to obtain this flexibility most of the capabilities of a general purpose programming language must be provided. This means that a relatively sophisticated language parser must be developed. As a third alternative, a language like PostScript or Forth might be employed in which there is relatively little syntax structure and thus, does not need a sophisticated parser. These languages are also extensible providing for even greater flexibility. However, these languages have a disadvantage in that they are often hard to read due to their use of postfix notation and of stacks. While these disadvantages can not be entirely eliminated, with some suitable extensions to the language and some discipline in writing input they can

¹PostScript is a registered trademark of Adobe Systems, Inc.

be reduced. GPS makes use of this last alternative.

The next section presents an overview of the GPS language and some of its differences with PostScript. The third section presents a brief discussion of the PostScript like language operators followed by a section on the new operators for dealing with models developed in C.

1.2. GPS Overview

The GPS language is structured exactly as the PostScript language with the few differences discussed below. Thus, [] delimit arrays, { } delimit procedures, and a / delimits literals. A % sign marks the beginning of a comment, which is terminated by a newline mark. GPS makes use of dictionaries and a dictionary stack as with PostScript, with systemdict containing all of the builtin operators and userdict, the first of any user defined dictionaries. These two dictionaries can not be removed from the dictionary stack. The dictionary stack can hold up to 20 dictionaries, the operand stack up to 500 elements, and the execution stack up to 250 elements. The userdict is initially defined to hold up to 200 user-defined entries. In GPS, however, no new words can be added to the systemdict.

At present, GPS implements roughly 60 of the PostScript operators and, in addition, about a dozen other operators for dealing with model classes and the mathematical utilities currently within SALT. The full list of PostScript like operators is given in the next section. Here we present some of the differences between GPS and PostScript.

The first major difference between GPS and PostScript is that many of the operators within PostScript have not been implemented simply because they were not needed for the purposes for which GPS is envisioned. These include all of the imaging operators, such as lineto, moveto, stroke, show, etc., many of the file operators, and the error handling operators, such as stopped, stop, etc. Some of these, notably the error handling operators, may eventually be added to GPS. Note the intention of GPS is not to compete with PostScript but rather to have a way of interacting with the SALT code models in an interpreted way while still maintaining the full capabilities afforded by an actual C++ driver. Secondly, character strings within GPS are delimited by double quotes " " as in C or C++. This frees up the use of parenthesis for another purpose which is the third main difference. Any collection of words that define an algebraic expression may be enclosed within parenthesis, in which case the expression can be written in the usual infix notation. Thus, the expression

$$((x+y)*\sin(\ln(x)+1)-2*\cos(x*y))$$

can be written exactly as is rather than

$$x\ y\ add\ x\ \ln\ 1\ add\ \sin\ mul\ 2\ x\ y\ mul\ \cos\ mul\ sub$$

as in PostScript. This can greatly improve the readability of the language when used to define algebraic expressions.

Another difference between GPS and PostScript is that GPS keeps a reference count of all arrays, procedures, and dictionaries, objects that use what PostScript calls virtual memory. These objects are allocated in memory and are pointed to by possibly many other objects. The number of objects that have a reference to one of these virtual memory objects is kept track of and stored with the object. When this reference count becomes zero the memory occupied by the object is freed. Thus, the GPS code automatically performs its own garbage collection and the usual save and restore type operations used in PostScript are not used.

Finally, there are a few minor differences between the GPS operators and the PostScript operators. These differences will be discussed within of the next section.

1.3. PostScript like operators

As many of the GPS operators are exactly like their PostScript counterparts, this section simply enumerates these operators, referring the reader to the PostScript Language Reference manual [3] for their detailed usage. These operators are listed in Table 1 which is followed by a very brief introduction to the use of these PostScript like operators.

The best way to understand the various GPS operators is to see how they are used in some examples. Thus, we present some very simple examples making use of only those operators that occur in Table 1 in an interactive GPS session.

GPS is started as an interactive session by simply typing GPS. The "gps>" prompt will then appear and any valid GPS input can be input in a free form way. Alternatively, the input can be typed into a file, edited, and saved, then a GPS session can be started and the GPS run operator used to simply execute the file. That is, after the "gps>" prompt, simply type

"file name" run

Note the run operator can also be used to initialize the system by simply executing a file containing any user defined words, dictionaries, abbreviations, or whatever. To terminate the GPS session, simply type the quit operator.

Table 1. GPS operators similar to PostScript

abs	add	aload	and
array	astore	begin	ceiling
clear	cleartomark	copy	cos
count	countdictstack	counttomark	currentdict
def	dict	div	dup
end	eq	exch	exit
false	floor	for	forall
ge	get	gt	if
ifelse	index	le	length
ln	loop	log	lt
maxlength	mod	mul	ne
neg	not	or	pop
put	quit	repeat	roll
run	sin	sqrt	stack
store	sub	systemdict	true
type	userdict	=	

GPS also has an interrupt mode which is initiated by typing a Control-c character at the terminal. When this mode is in effect the prompt is changed to "gps_int>". Any GPS input is equally valid in this interrupt mode, including the quit operator which will terminate the session just as in the normal mode. To go back to the normal mode from the interrupt mode simply type resume.

When GPS is reading input from the standard input file any errors that are found are reported and the session continues. If an error does occur the operand stack should probably be cleared (i.e. simply type clear) or examined (i.e. type stack to print out the current stack) before more input is typed in. When GPS is reading from a file different from the standard input file any errors that occur are again reported, but in this case, the GPS session is terminated.

GPS like PostScript makes use of a postfix notation, similar to a "reverse-Polish" calculator and thus requires some getting use to. In general, numbers, character strings, and literals (words starting with a '/') are simply pushed onto an operand stack. Operators pop their arguments off of the stack, perform their function, and then push their results back onto the stack. Thus, the input

```
gps> 2 3 add =
```

would push 2, then 3 onto the stack, the add operator would then pop the 3 then the 2 from the stack, add them and push 5 back onto the stack. The = operator would then pop the 5 from the stack and display it on the console. Any of the arithmetic, logarithmic, exponentiation, and trigonometric operators can be used in a similar way. For example:

```
gps> 2 sin 3 * =
2.7279e+00
```

```
gps> 1 log =
0.0000e+00
```

```
gps> 1.0 1.0 atan2 =
```

7.8540e-01

Thus, GPS can be used as a calculator. Since algebraic expressions written in postfix notation are not real clear, expressions can also be written in the usual infix notation if they are inclosed in parenthesis. Thus the above examples could be written as follows.

```
gps> (2+3) =
5.0000e+00
```

```
gps> (3*sin(2)) =
2.7279e+00
```

```
gps> (log(1)) =
0.0000e+00
```

```
gps> (atan2(1.0,1.0)) =
7.8540e-01
```

Variables can be assigned a name and defined as a word in a dictionary. This is done by first putting the name of the word onto the stack as a literal (i.e. prefix by '/'), followed by the word's meaning, and then followed by the def operator. The def operator will pop the preceding two elements from the stack and place the definition into a dictionary. Thus, to define x to be 2.0 one would write

```
gps> /x 2.0 def
```

Actually, each such defined word appears within one or more dictionaries on a dictionary stack. The def operator places newly defined words into the topmost dictionary on this stack which is known as the currentdict. Once defined, the word can be used in any place that its defined meaning can be used. Thus, x, defined above as the number 2.0, can then be used in any algebraic expression. For example,

```
gps> (x*x-1) =
3.0000e+00
```

To redefine x to some new value simply reuse the def operator. Def always checks to see if a word already exists in the currentdict first before creating a new word. Thus,

```
gps> /x (x-1) def
```

would replace the definition of x to be the value of x-1. Often the value of a word to be defined will appear on the stack before the literal representation of the word. In these cases the exch operator is useful. For example,

```
gps> 12 /x exch def x =
1.2000e+01
```

The exch operator simply exchanges the top two stack elements. There are other stack manipulation operators - pop, dup, index, copy, roll, clear, cleartomark, =, and stack which all work like their PostScript counterparts. Briefly, pop deletes the top stack value, dup duplicates the top stack value and pushes it onto the stack, index takes a number from the stack, counts down the stack by that number, and then duplicates that stack element on top of the stack, copy takes a number from the stack and then duplicates that many stack elements on top of the stack, roll takes two numbers from the stack, the first the number of stack rotations and the second represents the number of stack elements to rotate. Here a rotation is a circular shifting of the top element to the bottom and each other element moving up one slot on the stack. When the number of stack rotations is negative the rotations are performed in the opposite direction with the bottom element moving to the top and each other element moving down one slot on the stack. Clear deletes the entire stack, cleartomark deletes down to a '[' mark, = pops the top stack element and displays it (as best as it can) on the console, and finally, stack displays the entire stack on the console while leaving the stack unchanged.

There is also another operator for defining words into a dictionary called store. Store differs from def in that if the word being defined does not appear within the currentdict, all other dictionaries on the dictionary stack are searched for the word. If found in any of the dictionaries it is replaced, otherwise, the word is added as a new definition within the currentdict.

The dictionary stack is completely separate from the operand stack and is used to define the context in which the user defined words are interpreted. A dictionary can be created by using the `dic` operator which takes an integer from the operand stack representing the number of words the dictionary is to hold. The dictionary itself is then left on the operand stack and can be given a name and stored in the current dict or any other dictionary. For example,

```
gps> /mydict 20 dict def
```

defines `mydict` to be a dictionary to hold 20 words. The dictionary stack itself can be manipulated by using the operators `begin` and `end`. `Begin` takes a dictionary from the operand stack and pushes it to the dictionary stack where it becomes the currentdict. `End` pops the topmost dictionary from the dictionary stack and makes the dictionary below it on the dictionary stack the currentdict. Thus,

```
mydict begin
...
end
```

can be used to define a local context with `mydict` as the currentdict. Note any newly defined words using `def` used between the `begin` and `end` would then be placed in `mydict`. Likewise any words referenced between the `begin` and `end` would be searched for within the dictionary stack starting with `mydict`. When GPS is started two dictionaries are on the dictionary stack, `systemdict` which actually holds all of the builtin operators, - add, sub, mul, div, sin, cos, etc., and `userdict` which is initially empty but can hold up to 200 words. These two dictionaries cannot be popped off of the dictionary stack using the `end` operator. The words `systemdict`, `userdict`, and `currentdict` are also operators which push onto the operand stack their corresponding dictionaries.

As the above tends to imply words can be defined to hold anything within the language, such as the `mydict` dictionary, and not just numbers. Thus,

```
gps> /y "this is a string" def
gps> /z {2 exch sin *} def
gps> /u [1 2 3 4] def
```

defines `y` to be a character string, `z` to be a procedure, and `u` to be an array. Procedures and arrays are composed of user defined words, literals, operators, and even other procedures and arrays. Procedures are delimited by curly braces and arrays by square brackets (similar to C). Both can be manipulated as a single stack element once their right delimiter, either `}` or `]` is encountered. The main difference between the two is that as procedures are input a deferred state of execution exists. That is, words used within the procedure, as it is being input, are not executed. The words are simply collected into a single operand stack element. Procedure execution only occurs when some other word explicitly executes the procedure such as when a named procedure is referenced or used in a flow control construct. For example, using the definition of `z` above, the input

```
gps> 10 z =
-1.0880e+00
```

would evaluate `2*sin(10)` and print the results. Note that, due to this deferred execution, procedures may contain words that have not yet been defined. However, all procedures that are executed will require that each word within them be defined, otherwise the undefined error will occur.

GPS has a number of execution flow control constructs, similar to the C language's `for`, `while`, `if`, and `if-else` statements. Very briefly, these constructs include the `if`, `ifelse`, `for`, `repeat`, `loop`, and `while` operators. The operand stack syntax for their use is defined as follows.

```
cond { ... } if
cond { ... } { ... } ifelse
start inc bound { ... } for
count { ... } repeat
{ ... } loop
{ ... } { ... } while
```

Here the `"{ ... }"` indicate an arbitrary GPS procedure and should be thought of as a single stack element, `"cond"` is either a true or false, `"count"`, `"start"`, `"inc"`, and `"bound"` are numbers. In each case the preceding stack elements before the operator are popped from the stack before the operators are executed. These operators work as follows. The `if` operator checks `"cond"`, if true it executes the procedure. The `ifelse` operator checks `"cond"`, if

true it executes the first procedure, if false it executes the second procedure. The **for** operator pushes onto the stack a value beginning with "start" and then executes the procedure. The value is then incremented by "inc", pushed onto the stack, and the procedure reexecuted. This continues until the value has been incremented beyond the "bound" value. The **repeat** operator executes the procedure "count" times. The **loop** operator executes the procedure indefinitely. Note in this case the builtin **exit** operator must be used to terminate the loop. The **exit** operator would be somewhere within the procedure, probably within an **if** or **ifelse** construct. **Exit** can also be used to prematurely terminate the **for**, **repeat**, and **while** loops also. The **while** operator is not similar to anything within the current PostScript language and will be discussed within the next section. One other looping operator exists that takes an array or dictionary as an argument in addition to a procedure, this is the **forall** operator,

```
[ ... ] { ... } forall
dict { ... } forall
```

Here each element within the array is, in turn, pushed onto the stack and the procedure is executed. In the dictionary case, each word and its meaning is, in turn, pushed onto the operand stack and the procedure is executed.

In forming the cond value used in the **if** or **ifelse** operators, most of the standard logical connectives are available - **or**, **and**, **eq**, **ne**, **lt**, **le**, **gt**, **ge**. Each of these take two stack values and return to the stack a true or false. Actually, GPS does not have the boolean type and true is really a 1 and false is a 0. All of the relational operators work with numerical data values and, at present, no checking that the operands are numerical is done. Like the algebraic expressions relational expressions can also be formed within parenthesis in an infix way. When this is done the usual C representation of these connectives should be used - "**||**", "**&&**", "**==**", "**!=**", "**<**", "**<=**", "**>**", and "**>=**". Thus,

```
(x>=y && x<=z)
```

in the C language would be written exactly the same way in GPS. However, the parsing of these expressions do not have the same precedence rules as in the C language. Thus, in forming expressions using both the relational operators and the algebraic operators additional parenthesis should be used to clearly reflect the intended precedence. The parser works on both the algebraic and relational expressions at the same time, in order to keep it simple, with the precedence of the "**>**", "**>=**", "**<**", etc. operators the same as "******" or "**/**" and the precedence of the "**&&**" and "**||**" operators the same as "**+**" or "**-**". The two operators **eq** and **ne** can also be used with literals and strings.

In addition to the **forall** operator for dealing with arrays, there are several others. Like the **dict** operator there is an **array** operator for creating arrays. **Array** takes a number from the stack representing the number of elements the array is to hold and then creates an array of that size. Thus,

```
gps> /myarray 10 array def
```

creates **myarray** as an array of ten elements. Arrays created this way are initially empty. Elements within the array may be assigned values using the **put** operator. **Put** takes an array, an array element index (the first element is 0 as in C), and a value from the stack and then redefines that element to be value. Similarly there is a **get** operator that takes an array and an array element index from the stack and then returns that element from the array to the stack. The **put** and **get** operators can also be used to put and get elements from a dictionary. In this case rather than an element index, a literal defining the dictionary word is used. In this regard, **put** is a third way, besides the **def** and **store** operators for defining words in a dictionary. At present, unlike PostScript, **put** and **get** cannot be used with strings.

Two other array operators are used to store and load (to the stack) whole arrays. The first is called **astore** which takes the top element from the stack which should be an array, determines its length, and then pops from the stack enough elements to fill the array. The array with its newly defined elements is then left on top of the stack. The second operator is called **aload** and takes the top-stack element which should be an array, and then pushes onto the stack each element of that array followed by the array itself.

At times it is useful to be able to determine the type of word that appears on the operator stack. The **type** operator can be used for this. **Type** takes an element from the stack and returns a literal with one of the following names - **numbertype**, **nametype**, **dicttype**, **arraytype**, **proctype**, **stringtype** or **unknown type**.

1.4. Additional GPS Operators

Some additional operators which are not defined in PostScript or are sufficiently different from those in PostScript are listed in Table 2. Some of the more complex of these are described in more detail below, others are simply abbreviations to the PostScript operators so that the usual infix notation can be exploited in parenthesized expressions.

The trigonometric functions in both Tables 1 and 2 take their arguments in radians rather than degrees as in PostScript. The additional trigonometric functions and the pow function in Table 2 all correspond to their C language counterpart, each popping the arguments that they need from the stack. For those functions requiring two arguments, the arguments are placed on the stack in the order that the C function requires. Thus, pow(x,y) in C would be x y pow in GPS.

Table 2. Additional GPS operators			
operator	meaning	operator	meaning
+	abbreviation for add	-	abbreviation for sub
*	abbreviation for mul	/	abbreviation for div
atan	arc tangent function	tan	tangent function
atan2	arc tangent function	asin	arc sin function
acos	arc cos function	bind	implements operator-binding
pow	C pow function	exp	C exp function
min	minimum of two values	max	maximum of two values
while	implements while loop	debug	turns on debugging
fopen	C-like fopen function	fclose	C-like fclose function
printf	C-like printf function	fprintf	C-like fprintf function
sintrp	signal-interrupt mode	resume	resume main mode
rangecheck	turns on rangechecking	estack	print execution stack

The while operator takes two procedures from the stack and executes the first. This procedure should return either a true or false value to the stack. The value is then popped from the stack and checked. If the value was true the second procedure is executed and the while loop executes the first procedure again repeating the process. If the value was false the loop is terminated. Note that the true or false returned by the first procedure is consumed by the while operator and is not on the stack when the second procedure is executed or when the loop terminates. However, one or both of the procedures may leave other values on the stack. This looping operator was furnished to give a mechanism similar to the C while loop. Basically, the C language construct

```
while (condition) { ... }
```

becomes

```
{condition} { ... } while
```

in GPS.

The fopen operator is similar to the PostScript file operator taking two character string arguments from the stack. The second argument popped from the stack is the name of the file and the first argument popped from the stack is the mode in which the file is opened. At present, only "w" for writing or "a" for appending are valid modes. The operator leaves on the stack a file descriptor. For example,

```
"file" "w" fopen
```

will open a file named "file" for writing, leaving the file descriptor on the stack.

The fclose operator takes a file descriptor argument from the stack and closes the associated file.

The printf operator takes either one or two arguments from the stack. If the top stack element is an array then printf pops an additional stack element which should be a character string argument representing the format control of a C-like printf function. Like the C format control string, instances of % followed by the usual format control characters, i.e. e, d, f, or s are recognized. In addition the usual escape sequence of '\n'

corresponds to a new line as in C. For each occurrence of a %, a corresponding element of the array argument will be printed. For example, to print the variables x, y, z with the C format string

```
"x=%.2f y=%5.4e z=%d"
```

one would write

```
"x=%.2f y=%5.4e z=%d" [x y z] printf
```

At present only the format controls e, f, d, and s are recognized. **Printf** can also be called with just a format string as an argument if no variables need to be printed.

The **fprintf** operator is exactly like the **printf** operator except that an additional argument is required on the stack before the format string representing a file descriptor obtained from the **fopen** operator. For example,

```
fd "nx=%e" [x] fprintf
```

will print the value of x, labeled by "x=", onto the file associated with the file descriptor, fd.

The **bind** operator is used to bind all of the builtin operator names that appear within a procedure to the operations themselves. This prevents looking up the operators within **systemdict** during the execution of a procedure thus speeding up its execution. **Bind** takes either an integer or a procedure from the stack. If its argument is a procedure, **bind** performs its function on that procedure (but excluding any nested procedure) and returns the procedure to the stack. If its argument is a number, it should be either a zero or one. A one indicates that an autobinding should take place for every procedure as it is defined. A zero turns off this autobinding mechanism. Initially autobinding is on. Additionally, binding resolves any variable defined in the model classes to a pointer to the variable.

The **debug** operator is used to set the level of debugging. **Debug** takes the top stack argument as the appropriate debugging level. There are 5 levels of debugging. Zero turns off any debugging. A one prints out the top five stack elements followed by the current word being executed. These words are preceded by the word stack and three comma separated numbers. The first number is the top stack element number, the second is the top execution stack element number and the third is the top dictionary stack element number. A two level debug gives the same as level one plus also shows the elements for each array or procedure. A three gives the same as the two level plus shows each word as it is collected into procedures. Finally, a four gives the same as level three plus, shows each model class variable that is bound to its location and shows the virtual memory counts for each array, procedure, or dictionary as well as their hexadecimal locations. As each word is shown, either in executing, collecting, or binding, both the name (if known) and the type are shown with the type printed first followed by the name separated by a colon. In the case of arrays, procedures, or dictionaries, the type is followed by the current reference count for the object separated by a colon. There are a number of type letters that can presently appear. These are shown in Table 3. Initially debugging is off.

The **sintrp** operator is used to take the GPS interpreter into an interrupted mode. When this operator is called GPS suspends its current execution and prompts for additional GPS input from the console. To distinguish this mode from the normal mode, the prompt is changed to "gps_int>". Any legitimate GPS input can

Table 3. Word types

type	meaning	type	meaning
0	type not yet set	L	literal
S	character string	N	number
D	dictionary	A	array
P	procedure	O	builtin operator
I	infix parenthesis expression	F	file descriptor
d	model class double variable	i	model class int variable
s	model class char* variable	e	model class double and literal
j	model class int and literal	t	model class char* and literal
g	model class function and literal	f	model class function

then be entered. This input is kept completely separate from the suspended work in so far as the operand stack is concerned. However, all the user defined words, dictionaries, model classes, etc. used by the suspended work can still be used. Thus, `sintrp` can be used to interrupt the execution of some GPS input for examining variables or even reassigning different values to variables. For instance, the `debug` operator could be turned on or off. However, in changing variables that are being used in some suspended work, one should be careful that such changes make sense. Thus, the use of `sintrp` should be carefully thought out before the GPS input is executed. An example of the use of `sintrp` will be shown in the next chapter.

At times it is useful to be able to interrupt the execution of some GPS input, not from within the GPS input itself, but from the keyboard. This is done by simply typing Control-c. When this interrupt signal is caught by the GPS interpreter, the interrupt mode will commence.

The `resume` operator is used to resume the normal processing mode when in the interrupt mode. The processing mode that is resumed is exactly that which was executing before the interrupt. Thus, if a file was being executed with the `run` operator, that file execution is resumed.

The `rangecheck` operator is used to suspend checking of the array bounds when using the `put` or `get` operators with arrays. The operator takes one argument from the stack which should be either a 0 for turning off rangechecking or 1 for turning on rangechecking. Note, that like many other codes, if rangechecking is turned off and the bounds are exceeded, a segmentation fault will probably occur. Thus, at least initially, rangechecking should probably be kept on. The on state is the default. The ability to turn off rangechecking is provided to speed up execution of those inputs making use of many large arrays.

The `estack` operator is similar to the `stack` operator only it prints out the state of the current execution stack.

Several additional operators are also furnished for dealing with model classes and the mathematical utilities classes. These operators are listed in Table 4. Examples of their usage are presented in the next chapter. A model class is a C data structure and a collection of functions that take a pointer to that structure as an argument. Thus, a GPS model class is similar to a C++ class. An instance of a model class is simply an allocation of the C structure. In general once a model class instance has been allocated, its variables can be used just like any other variable within the GPS input. However, in order that GPS can determine that these variables are model class variables a restriction is placed on the names that variables can have in the GPS input. Variables that are not model class instances must never have a "." embedded within their names. Variables that are model class instances are referenced exactly as they would be in C, as the instance name, followed by ".", followed by the variable name. The additional operators will now be explained.

`cinit` - `cinit` is used to initialize any mechanisms needed by the model classes before any such class is defined. This operator should be called once before any model class is used (i.e. such as with `cnew`). `Cinit` requires no stack values as arguments and returns no stack values.

`cnew` - `cnew` is used to allocate a new model class instance and store it on a special stack for use in referencing its member functions and variables. `Cnew` requires one array object on the stack, the elements of

Table 4. Operators for handling GPS model classes	
operator	meaning
<code>cinit</code>	initializes model class interface mechanism
<code>cnew</code>	allocates a new model class instance
<code>cdel</code>	deletes a model class instance
<code>call</code>	calls a model class member function
<code>vary</code>	defines variables to be varied
<code>cons</code>	defines equality constraints
<code>icons</code>	defines inequality constraints
<code>mini</code>	defines objective functions for optimizations
<code>diff</code>	defines differential equations

which are a literal specifying the class type, a literal specifying the name of the class instance and zero, one, or more pairs of elements consisting of a class variable name specified as a literal and a value for that variable. For example, to allocate a new instance, denoted x, of the model class abc, and to assign values 1, 2, and "abc" to the class members parm1, parm2, parm3, one would write

```
[ /abc /x /parm1 1 /parm2 2 /parm3 "abc" ] cnew
```

Cnew returns no stack values. Note that within the array argument to cnew the model class variables are referred to without being qualified by the instance name and thus, do not contain the embedded '.'. Note also, that class abc may have many other variables besides parm1, parm2, and parm3. Not all of the class variables have to be initialized with cnew. Additionally, class variables do not have to explicitly appear within a cnew operation in order that they may be referenced later. Thus, once a class instance is allocated with cnew all the variables of the class can be referenced.

cdel - cdel is used to free up any model class instance allocated by the cnew operator. It requires one stack value which is the name of the class instance used in the cnew operator. Thus, to free the instance x of class abc defined in the previous example, one would write

```
/x cdel
```

Cdel returns no stack values. As it is sometimes necessary to free all of the variables allocated by all of the cnew operations, the special literal value /all is recognized by cdel. In this case all of the model classes will be freed. Once a class is freed any reference to it will be flagged as an undefined error. Cdel used with the /all argument will also free variables that might have been allocated with cinit. Thus, cinit would have to be recalled to make use of further model classes in the same input.

call - call is used to reference a model class function which takes arguments. In general, a model class function can be called simply by specifying its name. For example, if model abc had a function c taking no arguments, then to call the x instance of that function one would simply write x.c in the GPS input. However, suppose the model also had a function d taking 4 arguments then the call operator should be used. The call operator takes from the stack an array of values representing the arguments in the order that the function requires and the name of the function specified as a literal. For instance

```
[ 1 2 3 4 ] /x.d call
```

Note that call does not return any stack values, however the function being called may return an integer, double, or character string, which will be simply pushed onto the stack. At present, only functions with numerical and string arguments may be called.

vary - vary calls the C function necessary for solving systems of algebraic equations and/or differential equations. When used in solving algebraic equations vary requires four stack values consisting of a variable to be varied specified as a literal, a starting value, a lower bound value and an upper bound value. When used with differential equations vary requires two stack values consisting of the dependent variable of the differential equation specified as a literal and the initial value of that variable. Vary does not return any values to the stack.

cons - cons calls the C function used for defining and storing system constraints. Cons requires two stack values consisting of a variable specified as a literal (this is used to delimit this constraint from others, that is, as a constraint label) and the current value of the constraint residual. Cons returns no values to the stack.

icons - icons calls the C function for defining inequality constraints and works exactly as cons, requiring two stack values.

mini - mini call the C function for defining and storing objective functions used in optimizations and requires one stack value representing the current value of the objective function. Mini does not return any values to the stack.

diff - diff calls the C function for defining differential equations and requires two stack values. The first is the name of the variable to be integrated written as a literal followed by the current value of the variable's derivative.

Chapter 2 will explain the use of the vary, cons, icons, mini, and diff operators in more detail, along with examples of their use.

CHAPTER 2

General Model and Mathematical Utilities Classes

2.1. Introduction

The GPS code was designed to analyze systems consisting of arbitrary lumped component models interconnected by various "flows". Most often these "flows" represent actual physical flows, such as gasses or liquids, although they may just represent information that needs to be passed from one model to another. The models represent discrete entities through which the flows pass. In physical systems these entities are the pumps, compressors, turbines, nozzles, diffusers, heat exchangers, reactors, mixers, splitters, etc., that make up the system. The interconnectivity of the models by the flows represents the system configuration. GPS was designed to handle an arbitrary system configuration. In addition, GPS was designed to let the user impose on the system arbitrary system constraints, parameter sweeps, optimization studies, etc. In this chapter we consider the general features of the model and utility classes. Later chapters will discuss specific model classes for handling power systems and thermionic systems. There are actually several different collections of model classes with each collection being stored in a different library of models.

The next section describes some general conventions used by the different classes followed by a section on the use of stacks. The fourth section discusses the task class and its member functions. This class is one of the most important classes in that it is the controlling mechanism in solving most system analysis problems. This section is followed by a section giving a series of examples of the use of the task class.

2.2. Class conventions

A model or utility class in GPS is nothing more than C data structure and a collection of functions. As mentioned briefly in chapter one, the C variables within the data structure are referenced within the GPS input as the class instance name, followed by a ".", and then the variable name, that is, they are referenced exactly as they would be in C. The function names are also referenced within the GPS input in the same way, as the class instance name, followed by a ".", and then the name defining the function. We will often refer to the functions associated with a class as member functions analogously to the terminology used in C++.

The elements of a model class structure may, themselves, be instances of other classes, i.e. substructures. For example the substructure holding the values of the model powers and flow variables are treated like model classes. Thus, there really is no limitations on what can be placed within the model class structure. For uniformity, the actual C member functions of the class are usually named the same as the data structure followed by a suffix delimiting the specific function. Usually, the first argument to these functions is a pointer to the data structure.

In general, the model classes each have an allocator function denoted by the suffix "new" which is used to allocate an instance of the class and to define the default values to the model data structure members. This new function requires a character string argument which represents the class instance name and is stored in a variable denoted as "name". In addition, this function will usually place the model onto a stack, which can then be used to manipulate all of the models as a unit. All of the models have a calculational function, usually denoted by the suffix c, and a printout function, denoted by the suffix print. The calculational function is where the actual model equations are solved. In addition, each of the models has a ref function for referencing the model variables by name. This ref function is the communicating link between the GPS code and the model variables and will be explained in more detail in a later chapter.

2.3. C Stacks

The model and utility classes make much use of various stacks for storing and retrieving information. These stacks are themselves composed of a data structure and member functions and thus, are themselves model classes as we are defining these. These stacks are also completely different than the stacks used in GPS. The

GPS stacks are completely under the control of the user. The stacks used by the model and utility classes are only controlled by the user in an indirect way. Thus, as mentioned above, when a new model instance is allocated it is usually placed on a stack. Whether or not it actually is is dependent on the specific model. When the user defines system constraints, as will be discussed later, these constraints are also placed onto a stack. Other stacks are used to hold the flows that are passed from model to model. Some of these stacks have specific names that can be referred to within the GPS inputs. For example the stack upon which the models are placed is denoted as "mods". This stack has a print function referenced as "mods.print", which when called will, in turn, call the print functions of all the models on the mods stack. Similarly, the stack used to hold the gas flows in the power system component library is denoted as "gass". A similar print function, "gass.print", when called will produce tables of the all the gas flows within the system. Each model class library will, in general, have one or more of these stacks that need to be present in order that the models will work properly. The cinit function is used to allocate and properly set up any such stacks required by the models.

2.4. Task class

The task class is used to set up and control the iterations used by the mathematical utilities. The current set of utilities includes a hybrid steepest descent/quasi Newton update technique for solving systems of nonlinear algebraic equations [4], a sequential quadratic programming technique for solving nonlinear constrained optimization problems [5], and Gear's method for solving systems of stiff and nonstiff ordinary differential equations [6]. The task class makes use of some member functions to collect into separate stacks the problem data for the particular task being solved. These include the variables being varied using vary, the equality constraints using cons, the inequality constraints using icons, the objective functions using mini, and differential equations using diff. When the controlling function of the task, denoted as c, is called, it determines the type of problem that has been set up, allocates the appropriate work space, and then calls the appropriate mathematical utility. While the details of the equation solvers, optimizers, and ODE solvers is beyond the scope of this document, the task variables should be understood to effectively use the task class within the GPS input.

The complete list of user variables for the task class is given below. For each variable an indication of whether the variable is an input or output is given, along with its default value specified in parenthesis.

- maxit - integer defining the maximum number of iterations that are allowed in solving equations and in performing optimizations (40). Input. Maxit should be less than 1000, as iteration counts greater than that have a special meaning the the equation solver and optimizer.
- prt - integer specifying various amounts of output be printed during the iterations that the task is performing (2). Input. The value zero will turn off all printing requiring that any output be generated explicitly by the GPS input. Values greater than zero will produce greater and greater amounts of output. The actual output that is generated is dependent on the task being solved and also requires for its interpretation a greater understanding of the mathematical utilities than can be quickly explained here. However, the default value of 2 provides for a reasonable amount of output for most tasks and, as this is the default, this level of output will be explained.

For the equation solving tasks the following is obtained. For each iteration, the output will consist of the task name (as furnished by the user within the GPS input) labeled as (task=), the iteration number labeled as (n=), and the square root of the sum of the squares of the constraint residuals labeled as (f=). Note that this last value should gradually be reduced to zero as the iterations proceed. Following these values is the list of independent variable values, i.e. the unknowns of the problem, labeled as (x=) and the list of constraint equation residuals labeled as (c=). This last list of numbers should also gradually be reduced to zero as the iterations proceed. Following these items is a line of output giving some values of Newton step norms, steepest descent step norms, etc. Only one of these will be important in most cases, and that is the variable labeled as (mu=). This variable gives some measure of the ratio of Newton step versus steepest descent step and will generally be a small number, (less than about 3) if the equations solver is not having problems. If mu becomes larger (greater than 10) then one should reconsider the problem being solved. For example, it might be singular or not even have a solution.

For the optimization tasks, the outputs give the task name (task:) and the iteration number (it). The number of equality constraints (meq=) and the objective function value (f=) are then given. The next line (x=) give the values of the independent variables. The (c=) line then gives the values for the constraints with the equality constraints specified first, followed by the inequality constraints. Note that unlike the equation solver tasks, the number of independent variables and constraints may be different. A line labeled as (l=) gives the value of the terminations function (a function similar to the gradient of the Lagrangian only with absolute values within its sums). When this value is less than the specified task accuracy the problem is considered solved. The value of l is only calculated after a quadratic subproblem has been solved and thus, does not appear on every iteration. Some of the iterations are line searches which will include a output line that gives the number of the line searches (nfl=) plus several other parameters pertinent to the line search.

For integration tasks, again the task name labeled as (task:) is given followed by the current time (t=), the integrator state (state=) and integration order (order=). The next line labeled as (x=) gives the dependent variable values and the last line gives the dependent variable derivatives (dxdt=).

- acc - variable holding the termination accuracy criteria(1e-3). Input. For equation solving tasks when ever the square root of the sum of the squares of the constraint residuals becomes less than acc the iterations are terminated.
- del - variable indicating the amount of perturbation that the independent variables will undergo when the equation solver or optimizer is calculating gradients of the constraints (1e-7). Input.
- meth - method used by the ODE integrator indicating whether Gear's backward differencing method if 1 or the Adams-Bashford-Moulton method if 0 is to be used (1). Input.
- state - variable indicating the state of the ODE integrator(0). Input. Initially this variable is 0 indicating to the integrator to start the integration. On output it is assigned a value from 1 to 7 indicating the type of step that the integrator is performing. This variable should be manually reset to zero at the start of an integration task if one is performing an iterative loop around such a task. State values of 1 indicate that the integrator has reached a specified output time. State values of 2 indicate that the integrator has reached a time value for which the dependent variables are known to the requested accuracy. These two values of state are the only ones for which it is guaranteed that the time values reached will not become smaller. For all other state values, the integrator may be performing iterations, jacobian evaluations, or other functions for which a later step might actually be done for an earlier time value. This would be the case, for instance, if the integrator could not maintain the requested accuracy for the current integration step and had to reduce it. This is mentioned because it is often desirable to print out some variables while an integration is being performed, and it is only when state is 1 or 2 that the print out of such variables would make sense.
- time - variable indicating the present value that the variable being integrated over has reached (0). Input on the first call. On output time will contain the current time reached during the integration. This variable should also be manually reset if the integration task is repeated within some iterative loop. Note that this variable is denoted as time since very often time is the independent variable for the integration. This, however, does not preclude using the integrator for integrating over other variables, they must just be denoted as time.
- tout - variable indicating the output value to which the integrations will continue (1.0). Input. If several output times are required, the integration task should simply be put within an iterative loop over tout. Note that this loop does not repeat the integrations from their start so time and state should not be reset to zero in this case.

Class **task** also has five primary member functions, vary, cons, icons, mini, and diff. (Note, that these functions are used so often that the normal class naming convention of prefixing them with "task" is not used.) These, as briefly explained within the introduction, are used to setup the various task types. Each of these functions should lie within a loop controlled by the c function of the task. This controlling function should be called before any of the member functions are called and returns a one if the task is not yet satisfied (i.e. equations not

yet solved or integration output time not yet reached) and a zero when the task is satisfied. The easiest way to use this function within the GPS input is to place the `c` call within a while statement:

```
(x,c)
{
  "task body"
}
while
```

where `x` refers to the actual task object that has been declared for this task and "task body" will define the problem to be solved using the `vary`, `cons`, `mini`, etc. GPS operators.

The first task class member function, `vary` requires five arguments. The first is the address of the variable being varied, the second is an expression of the starting value for this variable, the third and fourth are expressions for the lower and upper bounds between which the variable will be constrained to lie. Initially, the variable should be between these bounds. The fifth argument is the address of the specific task instance that this variable is associated with.

The second member function, `cons` is used to define algebraic constraints or equations that need to be solved. This function requires three arguments. The first is used only to reference the constraint and is the address of any variable that remains in existence during the life of the task's controlling loop and that is not used within any other `cons` function call. Typically, one would use the address of one of the variables being varied within a `vary` call. If necessary, one could define a dummy variable and use that for this first argument. Note, it need not have a value or even any meaning for the problem. Its only purpose is so that each time the `cons` function is called this variable can be checked to see what constraint is being defined. The second argument is an expression representing the equation to be solved. At the solution this expression should become zero (to within a specified accuracy). The third argument is the same as with the `vary` function and is used to refer to a specific task that this constraint is associated with.

The next member function `icons` is exactly as the `cons` function, but is used to define inequality constraints. This function would only be used when one is defining an optimization problem. Here, the second argument at the solution will be constrained to be greater than or equal to zero.

The `mini` function is used to define objection functions for optimization problems. It requires two arguments, the second of which again refers to a specific task. The first argument is an expression representing the objective function for the optimization task. At the solution this first argument should represent a local minimum of the objective function.

The last member function, `diff` is used to define ordinary differential equations for the task. If this function is called, then `cons`, `icons`, and `mini` should not be called for this task. It requires three arguments, the last being the address of the task instance as with the other functions. The first argument is the address of the dependent variable for the differential equation being defined. This equation is of the form

$$\frac{dx}{dt} = f(x, t).$$

Thus, the first argument would be the `x` in this equation. The second argument is the expression `f`. As noted above, the variable `t` is represented by the class variable, `time`.

As mentioned previously, these task class functions are called within the GPS input by using operators with the same names, that is, `vary`, `cons`, `icons`, `mini` and `diff`. In this case, those arguments that were required as addresses are specified as literals within the GPS inputs. In addition the last argument to each function is automatically filled in by the GPS coding. These functions are also sometimes used directly within some of the C model classes.

There are also several other task class member functions `varyl`, `consl`, `iconsl`, and `diff1` which are exactly like those described above except that a character string rather than a pointer is used for the first argument. These functions are used only within the GPS code itself. Finally, one other member function `diffv` is used to assign initial values to any variables being integrated. This function is also only used within the GPS code.

2.5. Task Class Examples

In this section several examples are presented that make use of the task class. The examples presented should give a flavor of the type of problems that can be set up and solved by showing how to solve purely mathematical problems, such as solving equations, performing optimizations, and solving systems of differential equations. These basic techniques will then be used in later chapters with actual systems models to form and solve system constraints, optimizations, etc. Appendix A shows each of the following examples with their resulting output.

2.5.1. Example one

The first example sets up a purely mathematical problem of solving a single equation in a single unknown. The equation is

$$x^2 - e^{-x} = 0.$$

Problems such as this are solved by varying the value of x iteratively until the equation is satisfied. Thus there are three major aspects to solving the problem. First some iterative loop must be defined. This loop will be called the task loop; the task, in this case, is to solve the equation. The task loop will need to control the iterations and to terminate when the task is solved. The second aspect is to define what variable is being varied to carry out this task and to define a starting value and bounds for this variable. The third aspect is to define the equation to be solved. This equation will also be called the constraint for the task. In order to easily specify each of these aspects some simple operators have been created. Thus, in order to specify the variable to be varied the vary operator is used. For specifying the constraint equation, the cons operator is used. For defining task control, a task class instance is allocated using cnew and some GPS loop construct (e.g. the while operator) is used to define the iterative task loop. The complete GPS input necessary to solve the problem is as follows.

```
cinit
[/task /a] cnew
{a.c}
{/x 1.0 0.0 2.0 vary
 /x (x*x-exp(-x)) cons
}
while
/a cdel
```

Here the cinit operator is called to perform any model class initiation. This must always be done before any other reference to a model class, including the task class is made. The cnew operator is then used to define the task instance denoted as a. In general, the task controlling function is given by the task member function named c. Thus, in this case the task controlling function would be referenced within the GPS input as a.c. An iterative while loop is then started to carry out the computations within the task. In this case the task controlling function a.c is called which will return a 1 (true) until convergence is obtained, thus causing the second procedure defining the varying of the x and evaluation of the constraint equation to be executed iteratively. When convergence has been obtained the a.c function will return a 0 and the while loop will terminate. As indicated previously, the vary operator takes the name of the variable to be varied specified as a literal, in this case /x, followed by a starting value, and lower and upper bounds, here taken as 1.0, 0.0, and 2.0, respectively. The cons operator takes a literal (for labeling the constraint), here specified as /x, and the equation residual. The last line in the GPS input simply deletes the task a. Note that, since no parameters for the task a were initialized with the cnew operator, the default value for the task printout will be in effect and thus, the default level (a.prt=2) print out for this problem will appear on the standard output file.

2.5.2. Example two

The second example extends the first example to a system of algebraic equations to be solved. For illustrations, suppose these equations are

$$(x-1)^2 - y = 0$$

$$y - 2 \log(e^x + 1) = 0$$

$$z^2 - x = 0.$$

Here the GPS input would again consist of a single equation solving task but would include two additional vary and cons operators to define the two additional variables to be varied and the two additional equation residuals. Thus, the input is as follows.

```
cinit
[/task /a] cnew
{a.c}
  (/x 2 (-20) 20 vary
   /y 2 (-20) 20 vary
   /z 2 (-20) 20 vary
   /x (pow(x-1,2)-y) cons
   /y (y-2*log(exp(x)+1)) cons
   /z (z*z-x) cons
  )
while
  "\nx=%0.2f y=%0.2f z=%0.2f" [x y z] printf
/all cdel
```

As before one must decide on some reasonable starting values for x , y , and z and on the upper and lower bounds for these variables. At times this can be difficult and several different values may have to be tried in order to ultimately find a solution. This is especially true if the problem at hand has several solutions and one is seeking a particular one. In that case changing the bounds may be used to force the equation solver to search for a solution within a particular region. In this case, for lack of more information, the starting values for all three unknowns were taken as 2, and the upper and lower bounds taken as 20 and -20, respectively. Additionally, the `printf` operator was used to print out the final values (however, like the previous example, the default print out each iteration will also appear).

Since the task `a.acc` parameter defining the termination criteria was not specified, the default value was used stopping the iterations when the square root of the sum of the squares of the equation residuals was less than $1e-3$. This occurred on the 5th iteration where the sqrt of the sum of the squares of the residuals was $2.549055e-4$. If additional accuracy is required, `a.acc` should be made smaller. If substantially greater accuracy is required then for more difficult problems the default maximum number of allowed iterations, currently 40, defined by `a.maxit` will probably need to be made larger.

2.5.3. Example three

The third example sets up precisely the same problem as example two but in this case splits the problem into two-nested equation solving tasks. This is to show how complex problems might be decomposed into simpler tasks (although this example is easily solved as a single task). In this case, after calling `cinit`, two class task objects, `a` and `b`, are allocated with `cnew`, one for each of the two equation solving tasks. In the example, z will be solved for within the inner task denoted as `b` and x and y will be solved for within the outer task denoted as `a`. In order to reduce the number of iterations to solve the problem, z is given the initial value 2 using a `def` operator before entering the task loops. In this way z can be initialized to its current value each time the inner or `b` task loop is started. This z value will generally be better than simply taking z with some fixed starting value. The complete input would be as follows.

```
cinit
[/task /a ] cnew
[/task /b ] cnew
/z 2.0 def
{a.c}
  (/x 2.0 (-20) 20.0 vary
   /y 2.0 (-20) 20.0 vary
   {b.c}
     (/z 2.0 (-20) 20 vary
      /z (z*z-x) cons
     )
  )
```

```

while
/x (pow(x-1,2)-y) cons
/y (y-2*log(exp(x)+1)) cons
}
while
"\nx=%.2f y=%.2f z=%.2f" [x y z] printf
/all cdel

```

As can be seen by the input, the only change compared to example two is the nesting of the inner task while loop to solve the equation in z within the procedure used to solve for x and y . Decomposing a problem into nested problems such as this is often an effective means of solving a problem that seems to be intractable using only one task. Note, that if such a nesting is done, it often helps to keep the tolerance within the inner loops tighter than the outer loops. This is to prevent the inner iterations from washing out the effects of small perturbations of the outer loop variables when gradients of the constraints are being calculated.

2.5.4. Example four

As a fourth example we show how a nonlinear constrained optimization problem can be solved. The problem for illustrations is as follows,

$$\min (x-1)^2+(y-2)^2+ze^z$$

$$\text{such that } x-y=0$$

$$x-z>0$$

and where all of the variables lie within 0 to 10.

Again a single task class can be used to solve the problem, in this case, using the icons and mini operators. The complete input to solve the problem is as follows.

```

cinit
[/task /a] cnew
{a.c}
{/x 1 0 10 vary
/y 2 0 10 vary
/z 3 0 10 vary
/x (x-y) cons
/y (x-z) icons
((x-1)*(x-1)+(y-2)*(y-2)+z*exp(z)) mini
}
while
"\nx=%.2f y=%.2f z=%.2f" [x y z] printf
/all cdel

```

Here, the starting values were taken as 1, 2, and 3 for the three variables. Like the cons operator, the icons operator takes a literal (used only to label or delimit this constraint from others) and the constraint residual. For inequality constraints this residual should be written such that it is greater than or equal to zero. Inequality constraints, of course, will not necessarily be zero at the solution, although they might be. For such optimization problems more inequality constraints can be imposed than the dimension of the problem. The mini operator is used to inform the optimizer what the objective function to be minimized is. Optimization problems are inherently more difficult to solve than purely equation solving problems; thus, at times one may need to redo the problem with different starting points and adjustments in some of the parameters used by the optimizer.

As with the decomposition used in the third example, additional nested tasks defining other optimizations or equation solvings can be included to define arbitrary problem types.

Although this problem is relatively easy to solve, with the final solution being obtained in ten iterations, this certainly is not always the case, and several points about optimization problems should be mentioned. First, such problems are considerably more difficult to solve than just solving algebraic equations. One cannot just look at the potential solution and "see" that it is the solution. This is because, looking at the residual to the

constraint equations and noting that the equality and inequality constraints are satisfied is only part of what needs to be considered. At the solution the Kuhn-Tucker conditions should hold. These conditions can only be evaluated by knowing the Lagrangian multipliers and gradients of the objective functions and constraints. Secondly, during the iterations it is quite possible that the value of the objective function may need to increase, for example, when one needs to go "uphill" in order to satisfy the constraints. Thirdly, iterative techniques like the one being used here, generally only find local minimums. To find a global minimum often requires substantially more work and sometimes requires apriori estimates of the second derivatives of the objective functions and constraints. These often are not available. Fourthly, the problem posed may not even have a local solution. This may occur, for example, when no feasible region exists for all of the inequality constraints taken together.

With these and other potential problems there are several termination messages that may occur when defining optimization tasks. The main ones are "initial line search gradient positive", "convergence of independent variables", and "more than 5 function calls in line search". Some of these may indicate that the solution was not found, while in other cases, they may signify that the solution was found but not to the level of accuracy requested. In some cases rerunning the problem from a different starting point can sometimes resolve the difficulty. At other times this may be the best that can be done with the finite differencing used in calculating the gradients. Sometimes a smaller (or even larger) value of del might be tried. Finally, one may have to decompose the problem, for example putting the equality constraints within an inner nested task or even resorting to parameter sweeps rather than an optimization task. Sometimes parameter sweeps will give greater insight into the problem under consideration and will indicate that some variables might be eliminated from the optimization problem, thus, reducing the dimensionality of the problem.

2.5.5. Example five

As a fifth example we set up an integration of three differential equations.

$$\frac{dx}{dt} = -x$$

$$\frac{dy}{dt} = \frac{y}{2}$$

$$\frac{dz}{dt} = x - y$$

Again cinit and cnew are used to define a task denoted as a. The default print out defined by the prt variable for the task is also set to 0 so that no print out will be generated. In order to generate several intermediate output values, a sweep is made on the variable defining the output times, denoted as a.tout, using a for operator. Since the for operator pushes onto the stack the current iterative value, the first thing done within the procedure is to use that value to redefine the current output value. Nested within this for loop is the task loop implemented using the while operator as before. Within the procedure of the while operator the three differential equations are defined using the vary operator to indicate the variables being integrated and to give these variables starting values and the diff operator for specifying the right-hand side of the differential equations. After the while operator the printf operator is used to print out the values of the time, and the three variables. The complete input is as follows.

```
cinit
[/task /a /prt 0] cnew
1.0 1.0 5.0
{a.tout exch def
{a.c}
{/x 1.0 vary /y 2.0 vary /z 0.0 vary
/x (-x) diff
/y (0.5*y) diff
/z (x-y) diff
}
while
"\ntime=%.2f x=%.3e y=%.3e z=%.3e" [a.time x y z] printf
}
```

```
for
/all cdel
```

Unlike the arbitrary nesting of equation solving and optimization tasks, differential equation solving tasks cannot be nested within each other. However, differential equation solving can be nested within or outside of equation solving or optimization tasks.

2.5.6. Example six

In this example, we consider the use of the `sintrp` operator. As described within the previous chapter, this operator signals an interrupt just as if the user had typed a Control-c at the keyboard. By using `sintrp` within the GPS input, the interrupt can be made to occur at exactly the right instance during the computations. Consider an example similar to the previous, only now let us suppose that after a predefined time, denoted as "trap" in the following, we want the integrations to stop and to go into the interrupt mode. In order to add, at least, one other parameter to the differential equations, the equations have been changed slightly with the parameter "p" introduced into the first equation. Initially "p" takes the value one and the first time trap is taken as one second. The inputs to accomplish this are as follows.

```
cinit
[/task /a /prt 0] cnew
/interup ("^ntime=%e" [a.time] printf sintrp) def
/trap 1.0 def
/p 1.0 def
1.0 1.0 10.0
{/a.tout exch def
{a.c}
{(a.state<=2 && a.time>=trap) (interup) if
/x 1.0 vary /y 2.0 vary /z 1.0 vary
/x (-x*p) diff
/y (0.5*y) diff
/z (x-y) diff
}
while
"^ntime=%.2f x=%.3e y=%.3e z=%.3e" [a.time x y z] printf
}
for
/all cdel
```

Here, as with example five the iterative task loop is set up defining the equations to be integrated, only now the first statement within that loop

```
(a.state<=2 && a.time>=trap) (interup) if
```

checks if the integration state is less than or equal 2 (see the discussion of the state variable in the task class) and also checks to see if the time is greater than or equal to the trap value. If these conditions are true then the procedure "interup" is called. Here "interup" was defined prior to entering the task loop to first print out the current time and then execute the `sintrp` operator. At that point, the GPS interrupt mode will become active and prompt for input at the terminal. Such input might simply be something such as

```
gps_int> x = y = p =
```

which would print out the values of x, y, and p. One might also redefine p to have a new value, such as

```
gps_int> /p 1.1 def
```

One must, however, before resuming, redefine the value of trap to be some later time, else on the very next iteration, the integrations will again immediately go back into interrupt mode. Thus, to resume the integrations and say, stop at time greater than or equal to 8.2, one would input

```
gps_int> /trap 8.2 def resume
```

When time becomes at least equal to 8.2 again the interrupt mode would be entered and variables can be

queried and/or changed as before. Note, that for this problem it would not make sense to change either x , y , or z , since they are the dependent variables of the problem. Also, as this problem indicates, some thought as to when the interrupt is to occur is usually required. In this case, it only makes sense when the integrator is in a state with a state less than or equal to 2.

When running a problem such as this, it is also possible to interrupt it from the keyboard with a control-c. In that case, variables can be queried, but the integrator state might not be appropriate for changing even an independent variable, such as the p variable in this problem. For instance, if p were changed while the integrator was trying to calculate the Jacobian of the right hand side of the equations, a bad Jacobian would result, possibly preventing the integrator from working. Thus, when problems are constructed to be interrupted at the keyboard, it is probably best to only adjust parameters within procedures that are called at appropriate times after the interruption is resumed.

CHAPTER 3

Steady-State Power System Model Classes

3.1. Introduction

In this section we discuss the details of the component models that are used to analysis a steady-state power system. These models consist of the following.

gas -	gas flow initiator
sp -	gas flow splitter
mx -	gas flow mixer
ht -	gas flow heater/cooler
hx -	gas flow heat exchanger
cp -	compressor
gt -	gas turbine
pump -	pump
df -	diffuser
nz -	nozzle
power -	calculate system powers

Each of these model classes have various parameters and member functions. For example, the hx model class has c and h functions to perform the calculations on the cold and hot sides of the heat exchanger. In general each model has several member functions, which in the following will be referred to by the suffix name only. Thus, the above hx class c function is actually encoded in C as the hxc() function. As discussed in the chapter on model and utility classes each of the models has a allocator function denoted by the suffix, new, requiring a character string as an argument. This character string will be used for referencing the model instance in any printout of the model's flows or parameters. The allocator function will also assign default values to any input parameters.

3.2. Steady-State Model Flow classes

Before discussing the model classes within the next section, some understanding of the flow classes required by these models is necessary. As indicated previously, the flow classes represent the information which passes between the different models. These classes will usually represent the variables describing real physical fluids, but can also represent most anything the modeler desires. In general, the user will manipulate the flows of a system by calling the models. In particular, for each flow class there is a special model that is used to initialize the flow and to save and restore the flow to a flow stack. This flow stack is unique for each flow class. Practically all of the models have as part of their class structure one or more instances of the flow classes. These are used to store the values of the flows at the exit of the model and can be used in forming constraints and/or objective functions within the GPS input.

The present list of steady-state models makes use of only one flow class, that of **gastype** for representing fluids and has the following variables.

id -	pointer to the flow's id
t -	flow's temperature in K
p -	flow's pressure in atm

h -	flow's enthalpy in J/kg
s -	flow's entropy in J/kg-K
r -	flow's density in kg/m ³
q -	flow's quality
m -	flow's mass flow rate in kg/s
v -	flow's velocity in m/s
atoms -	flow's atom fractions
comp -	flow's species mole fractions

Normally, the values of id, t, p, h, τ , r, q, m, and v will be the only variables that a user is likely to use. There are actually several different thermodynamic property codes available within the system. The actual procedure that is used to determine the properties is determined by the flow's identification pointer, id. This variable should be assigned a character string of the type "GAS" or "THR-species", where species is one of the several hundred species found in the THRDATA file. In the case where the pointer is to "GAS" the actual gas is further determined by the contents of the comp array. This array is dimensioned by the number of gas species defined in the prop.h file. For convenience, the species names (in caps) are defined as a sequence of integers so that the user can refer to a particular species by referencing its name. For example, the CO₂ mole fraction would be referenced as comp#CO₂. (Note that the usual way of referencing this array element in the C language would be comp[CO₂], however, the "[]" delimiters in GPS would define a GPS array rather than an element of the comp array. Thus, the usual way of referencing a C array element in GPS is by suffixing the array name with a "#" sign and then the element number. Note, however, that this referencing is actually defined not by the GPS code but, by the ref function to the class, and thus, could be changed by the user, if desired.)

In addition to the variables the gastype class has several member functions. These are generally only used within the model classes and thus, really don't need to be of any concern to the casual user, however they would be of concern to a model developer. Prop is the general property calculational procedure. For any instance of this gastype class, prop can be called to determine the thermodynamic properties of the flow either as a function of p and t, p and h, or p and s by using as its second argument the letter 't', 'h', or 's', respectively. Prop's first argument is the address of the flow.

Another member function, sat is used to determine the saturation properties of the flow at the flow's pressure. This function only returns values for flows with the "THR-species" id as the "GAS" flows are not condensible. If called with a "GAS" flow an error message is displayed and the run is terminated. Sat requires four arguments, the first is the address of the flow and the rest are double precision variables representing the returned values of critical pressure (atm), and the saturation liquid enthalpy and vapor enthalpy (J/kg).

The atom member function is only needed for flows with "GAS" as the id and is used to calculate the kg-atom/kg of the individual atoms making up the flow. This function requires one argument of the flow's address and uses the flow's comp array to determine the values of the flow's atoms array. Note that it is the atoms array that actually determines the chemical make up of the flow. This array remains constant until the flow either has new species added to it or removed from it. The flow's comp array, on the other hand, will change just like the flow's temperature or pressure and reflects only the current equilibrium species mole fractions.

One additional function is provided for use with the gastype class which is gasget and returns the next flow from the gass.

The special initializing model for this gastype flow is denoted as gas and will be described below. The unique stack for this flow is denoted as gass. The gass stack itself has several variables and two member print functions. These variables are as follows.

prt -	print flag (0). Input. Prt, when set to one, is used to print out values of the flow each time the properties code is called. Its use is really for debugging.
thrsat -	flag (1). Input. Thrsat, when set to one, will cause the THR properties code to first produce a table of the saturation temperatures as a function of the pressure. This table is then used to calculate the saturation temperature whenever it is needed by the THR properties routines. This is only a performance issue to eliminate the iterations needed to calculate the saturation

temperature later on. Note, however, these iterations must be done initially to generate the table.

quit - count of errors occurring within the property codes(5). Input. Quit is initialized to five and decremented on an error. When zero is reached it is assumed that the run is having too much difficulty and is terminated.

The two gass stack functions are the `print` function, which is used to print out tables of the flows' state variables, and the `printc` function, which is used to print out tables of the flows' species concentrations. `Printc` should only be used when one or more of the flows have the "GAS" flow id.

3.3. Steady-state models

The present collection of steady-state models do not have a lot of process related details but represent a thermodynamic description of the model's phenomena. This is basically the result of trying to keep a generic quality to the supplied models. New models with as much process detail as is required can be added by the user as will be discussed in a later chapter.

3.3.1. Gas (gas) model

The gas model is used to initiate a gas flow as well as providing member functions for performing the saving and restoring of flows for representing complex system configurations. The member function used to initiate a gastype flow is denoted as `c`. `c` requires no input flows and will generate one output flow put onto the gass stack. The modeling begins by simply assigning values to the flow variables as follows.

$$id=id_{in}$$

$$m=m_{in}$$

$$v=v_{in}$$

$$p=p_{in}$$

$$comp_i=comp_{i,in} \quad i=1 \dots NS$$

where id is the flow id as discussed above, m , v , and p are the flow's mass flow rate, velocity, and pressure, respectively, $comp_i$ is the flow's i -th species mole fraction and NS is the total number of species. The subscript in represents input values. Note, that NS is fixed by the property calculations procedures and is thus, not directly input.

The gastype's atom function is then called to determine the contents of the flows atom array. Note, that the assigning of species to the `comp` array and calling the atom function is really only needed for flows with the id of "GAS".

Next if the input value of temperature t_{in} is specified as zero, then the `sat` property function is called to determine the saturation liquid and vapor enthalpies, h_l and h_g . The flow's enthalpy, h is then determined from

$$h=h_l+q(h_g-h_l)$$

where q is an input value for the flow's quality. If the temperature, t_{in} is non-zero, then the flow's temperature is simply assigned this input value

$$t=t_{in}$$

and the `prop` function is then called to determine the flow's enthalpy. In either case, the `prop` function is again called with enthalpy as the input to determine the flow's density, entropy, etc.

The parameters to the model are as follows. The default values of the parameters are specified in parenthesis and an indication of whether the parameter is an input is also given.

id - gas flow id ("THR-tH2"). Input.

m - flow rate (1.0 kg/s). Input.

v -	flow velocity (10.0 m/s). Input.
p -	flow pressure (1.0 atm). Input.
t -	flow temperature (298.16 K). Input.
q -	flow quality (0.0). Input.
comp#i -	mole fraction of the i-th species. Input.
dp -	difference in pressure between the flow entering the cycl function (see below) that that leaving the c function.
dt -	similar to the dp variable but for temperature.
dh -	similar to the dp variable but for enthalpy.
dm -	similar to the dp variable but for mass flow rate.
dv -	similar to the dp variable but for velocity.
fl -	exit flow structure from the model. Note that fl needs to be further qualified with one of the gastype parameters, such as, "fl.t".

As noted above if the temperature is specified as zero then the model assumes that the flow is to start at the saturation temperature corresponding to the input pressure. In this case the specified flow quality is used to determine the inlet enthalpy. Thus, quality set to zero refers to the liquid saturation line and set to one, the vapor saturation line.

The member function used to save a flow, that is remove a flow from the gass stack, is denoted as sav. When a gas model instance is defined for use in saving (and recovering) a flow it will never be used in any printout. Also no input model parameters need to be specified.

The member function for recovering a saved flow is denoted as rec.

An addition member function denoted as cycl is also provided. This function requires one input flow from the gass stack and calculates the differences in temperature, pressure, enthalpy, mass flow, and velocity, denoted respectively by dt, dp, dh, dm, and dv, between this input flow and the output flow from the corresponding c function. This function is provided to help set up the system constraints on a flow path that forms a closed cycle. In addition, this function will calculate the difference in power (mass*enthalpy) between these two flows and save this in the variable power.heat. Note that for a correctly formulated closed path, this variable should be zero.

3.3.2. Mixer (mx) model

The mixer model is used to mix together two gastype flows using the member function c. This function requires one input flow to be on the gass stack and puts one output flow back onto the stack. The other input flow is obtained by calling another member function, s. This function, which must be called before the c function, requires one input flow on the gass stack but generates no output flows. The model requires no input parameters. Unlike the other models, since all the output is available within the gas flow outputs no print member function is used.

The modeling within the mixer is dependent on whether or not the input flows are "GAS" flows. For such flows, the output comp array of species mole fractions must be first calculated. This is done as follows.

$$mw_1 = \sum mw_i \text{ comp}_{1,i}$$

$$mw_2 = \sum mw_i \text{ comp}_{2,i}$$

$$\text{mol}_{1,i} = \text{comp}_{1,i} m_1 / mw_1 \quad i=1 \dots NS$$

$$\text{mol}_{2,i} = \text{comp}_{2,i} m_2 / mw_2 \quad i=1 \dots NS$$

$$\text{mol}_i = \text{mol}_{1,i} + \text{mol}_{2,i} \quad i=1 \dots NS$$

$$comp_i = m_i / \sum m_j \quad i=1 \cdots NS$$

where mol_i , $comp_i$, m , mw_i , and mw are the molar flow rate array, flow mole fraction array, flow mass flow rate, species molecular weight array, and flow molecular weight, respectively, and the subscripts 1 and 2 correspond to the two input flows. Once the mole fractions $comp_i$ of the output flow are known, the atom function is then called to determine the flow's atom fraction values.

For both "GAS" and non-"GAS" type flows the following calculations are then made to determine the output flow's pressure, enthalpy and mass flow rate.

$$p = \min(p_1, p_2)$$

$$h = (m_1 h_1 + m_2 h_2) / (m_1 + m_2)$$

$$m = m_1 + m_2$$

Finally, the prop function is called with enthalpy as an input to determine the flow's entropy, density, and temperature. At present, the mixer model cannot mix together flows with different flow id's.

The only output parameter for the model is

fl - representing the exit gastype flow from the model. As with all model flows, fl would need to be further qualified, such as fl.t when used within the GPS input.

3.3.3. Splitter (sp) model

The splitter model is used to split a gastype flow into two flows using the member function c. The function requires one input flow on the gass stack and will put one output flow back onto the stack. The second output flow can be obtained by calling the member function s. This function requires no input flows and should only be called after the primary function c is called.

The modeling done within the splitter is dependent on whether the splitter is being used to split off certain species or simply split the whole flow. If the split-ratio value, sr , is zero, then it is assumed that, at least, one element of the species split ratio array, ssr_i , is non-zero. In that case the mass flow rates of each species must be calculated to determine the split off flow. This is done as follows.

$$mw_{in} = \sum mw_i comp_{in,i}$$

$$m_{2,i} = ssr_i comp_{in,i} mw_i m_{in} / mw_{in}$$

$$m_{1,i} = (1 - ssr_i) comp_{in,i} mw_i m_{in} / mw_{in}$$

where mw_{in} is the inlet flow's molecular weight, $m_{1,i}$ and $m_{2,i}$ are the individual species mass flow rates of flow 1 and 2, $comp_{in}$ is the inlet flow's mole fraction array, mw_i are the individual species molecular weights and m_{in} is the inlet flow's mass flow rate. Once the individual species mass flow rates are known, the total flow rates for the two flows can be determined.

$$m_1 = \sum m_{1,i}$$

$$m_2 = \sum m_{2,i}$$

The new mole fraction arrays for each flow can then be determined as follows.

$$comp_{1,i} = m_{1,i} / mw_i$$

$$comp_{1,i} = comp_{1,i} / \sum comp_{1,i}$$

$$comp_{2,i} = m_{2,i} / mw_i$$

$$comp_{2,i} = comp_{2,i} / \sum comp_{2,i}$$

The atom function is then called for both flows to determine each flow's atom fraction array followed by a call

to the prop function with the inlet temperature as input to determine each flow's enthalpy, entropy, and density.

When sr is non-zero, the above calculations are replaced by the following.

$$m_2 = sr \cdot m_{in}$$

$$m_1 = m_{in} - m_2$$

In this case there is no need to call the prop function as the exit flow's state variables are the same as the inlet flow's.

The model's parameters are as follows.

- sr - split ratio representing the fraction of input mass flow rate that is split off to form the second output flow (0.5). Input.
- $ssr[i]$ - i -th species split ratio representing the fraction of the input mass flow rate of the i -th species that is split off to form the second output flow. Input.
- fl - primary flow structure from the model. Output.
- $fl2$ - secondary or split-off flow from the model. Output.

The ssr array may only be used with flows having the "GAS" id and is used only when the sr parameter is set to zero. Since both sr and ssr represent fractions of the input flow mass, their values should be between 0 and 1. The ssr array elements should not be all zeros or ones, as this would make one of the output flows have a zero mass. Note that since the sr variable is by default not zero, if ssr is to be used sr must be explicitly set to zero.

3.3.4. Heater (ht) model

The heater model is used to transfer heat into or out of a gastype flow. The model has one main calculation function c . The function takes one input flow from the gas stack and puts one output flow back onto the stack.

The model first calculates the exit flow pressure p based on an input pressure fraction f_p as follows.

$$p = p_{in} - f_p \cdot p_{in}$$

where p_{in} is the inlet flow pressure. The model then calculates the enthalpy change based on one of three options. If the exit flow temperature t is specified as non-zero, then the exit flow enthalpy is simply calculated from the prop function using t as input. If t is zero, then the model checks the exit flow quality, q and if that variable is greater than -100 then the sat function is used to determine the saturation liquid and vapor enthalpies, h_l and h_v at the exit flow pressure. These values are then used to calculate the exit flow enthalpy from

$$h = h_l + q \cdot (h_v - h_l).$$

Finally, if q is not greater than -100, the heat transferred, Q is used to determine the exit flow enthalpy from

$$h = h_{in} + Q / m$$

where h_{in} is the inlet flow enthalpy and m is the mass flow rate through the heater. Once the enthalpy of the flow is known, the prop function is called to determine the temperature, entropy, and density of the exit flow. In addition, the heat transferred from either the input or from

$$Q = (h - h_{in}) / m$$

is stored for later print out.

The parameters to the model are as follows.

- temp - temperature of the exiting gas flow (500 K). Input.
- qual - quality of the exiting gas flow (1000). Input.
- pfrac - fraction of the input pressure used as a pressure drop (0.0). Input.
- heat - heat input (w). Input.
- fl - exit flow from the model. Output.

Only one of temp, qual or heat should be input. Temp is used if not equal to zero. If temp is zero, then qual is used if greater than -100 (of course, if it is input it should be something reasonable). In this case the exit temperature will be the saturation temperature at the exit pressure if qual is between zero and one. Note that qual can be set less than zero to represent subcooled flow or greater than one for superheated flow. If either temp or qual is used to determine the exit flow temperature, then heat is an output variable. Finally, if temp or qual are not used (i.e. set to 0.0 and -1000, respectively), then heat is used directly to determine the exit temperature. Note that heat can be a negative number in which case this model will act like a flow cooler.

3.3.5. Gas turbine (gt) model

The gas turbine model represents a simple expansion to a given exit pressure at a given efficiency. The model has a main calculational member function denoted c. This function requires one input flow from gass stack and puts one output flow back onto the stack.

On entry to the model, the exit flow pressure, p is assigned the specified input value p_{exit}

$$p = p_{exit}$$

The prop function is then used with the inlet flow entropy as the input to determine the enthalpy h_s of the flow for an isentropic pressure change from the inlet to the exit. The exit flow enthalpy h is then determined from

$$h = h_{in} - \eta (h_{in} - h_s)$$

where η is the specified efficiency and h_{in} is the inlet flow enthalpy. The power produced is then calculated from

$$Pow = m (h_{in} - h)$$

Finally, the prop function is called with enthalpy as the input to determine the exit flow temperature, entropy, and density.

The model has the following parameters:

pres -	exit flow pressure (1 atm). Input.
eff -	efficiency of the expansion process (0.85). Input
power.work -	thermodynamic work generated by the expansion process. Output.
fl -	exit flow from the model. Output.

3.3.6. Gas compressor (cp) model

The gas compressor model represents a simple compression to a given exit pressure at a given efficiency. The model has one calculational member function denoted c. This function requires one input flow from the gass stack and puts one output flow back onto the stack. The compressor model is very similar to the gas turbine model only the exit flow enthalpy is calculated slightly differently. The model first assigns the exit flow pressure to be the specified exit value.

$$p = p_{exit}$$

Then the prop function is called to determine the enthalpy h_s of the isentropic compression to the exit pressure. The exit enthalpy h is then determined from

$$h = h_{in} + (h_s - h_{in}) / \eta$$

and the power required from

$$Pow = m (h - h_{in})$$

where h_{in} is the inlet flow enthalpy, m is the mass flow rate, and η is the specified efficiency.

The model has the following parameters:

pres -	exit flow pressure (5.0 atm). Input.
eff -	efficiency of the compression process (0.85). Input.

- power.work - thermodynamic work required by the compression process. Output. Note, that this parameter is treated as an algebraic quantity, with negative values indicating work consumed. Thus, in a normal compression process this parameter will be negative.
- fl - exit flow from the model. Output.

3.3.7. Heat exchanger (hx) model

The heat exchanger models the transfer of heat from a hot gastype flow to a cold gastype flow. This is done using two member functions *h* for the hot side and *c* for the cold side of the exchanger. Both of these member functions require one input flow from gass stack and put one output flow back onto the stack.

The model has several options and makes use of either a t_{cold} or a t_{hot} to specify the exit flow temperature t on either the cold or hot sides. The particular variable that is specified should refer to the function that is called first within the GPS inputs, either *h* or *c*. Thus, one has either

$$t = t_{cold}$$

$$Q = m (h - h_{in})$$

or

$$t = t_{hot}$$

$$Q = m (h_{in} - h)$$

where h_{in} is the inlet flow enthalpy on the appropriate hot or cold side and the exit flow enthalpy h is determined from a call to the prop function with the temperature as input. Once Q is known it is used on the other side to calculate the exit flow enthalpy, which, in turn, determines the other exit flow state properties using a call to the prop function. As an additional option, Q can be input directly rather than one of the exit temperatures. In this case, both t_{cold} and t_{hot} should be set to zero.

Once both sides of the heat exchanger have been called, the log mean temperature difference is calculated using stored values of the inlet and exit temperatures,

$$\Delta t_{mean} = (x - y) / \log(x/y)$$

where x and y are the inlet and exit fluid temperature differences of the heat exchanger. Note that for the purpose of using Δt_{mean} in system constraints, if either x or y or both become less than zero, a fictitious value of Δt_{mean} is returned, although one that still shows the correct trend as a function of x and y . Based on specified values of hot and cold heat transfer coefficients, u_{hot} and u_{cold} an overall heat transfer coefficient is determined from

$$u = \frac{1}{1/u_{hot} + 1/u_{cold}}$$

and the heat transfer area by

$$A = Q / (u \Delta t_{mean})$$

The model's input parameters are as follows.

- t_cold* - exit temperature (0.0 K) of the cold side.
- t_hot* - exit temperature (0.0 K) of the hot side.
- heat* - amount of heat (0.0 watts) transferred from the hot to the cold flows.
- ufh* - heat transfer film coefficient (1000 watts/m²K) for the hot side.
- ufc* - heat transfer film coefficient (1000 watts/m²K) for the cold side.
- type* - character string indicating the type of heat exchanger, "count" for counter flow or "paral" for parallel flow ("paral").

Only one of *t_cold*, *t_hot*, or *heat* should be input to the model. If either *t_cold* or *t_hot* is used then that side of the heat exchanger should be called first. These parameters are used to determine the value of heat which

then becomes an output parameter. If both t_{cold} and t_{hot} are zero, then the value of heat is used directly to determine the exit conditions.

The main outputs from the model are

- lmtd - log mean temperature difference across the exchanger.
- area - heat transfer surface area (sq. meters).
- flc - exit cold side flow.
- flh - exit hot side flow.

Note that if system constraints are to be placed on either lmtd or area, then the system task loop should include both the h and c functions for this model.

3.3.8. Pump (pump) model

The pump model represents a simple liquid flow compression process to a specified pressure at a specified efficiency. Note that this model assumes that the liquid is almost incompressible (constant density) and thus, should only be called where the flow is in the liquid region. The model has one calculational member function denoted c. This function requires one input flow from the gass stack and puts one output flow back onto the stack.

The modeling consists of the following equations.

$$Pow = m (p_{in} - p_{exit}) / (\rho \eta)$$

$$h = h_{in} - Pow / m$$

$$p = p_{exit}$$

where Pow is the power required, p_{in} is the inlet pressure, p_{exit} is the specified exit pressure, ρ is the fluid density, m is the mass flow rate, p is the exit flow pressure, h is the exit flow enthalpy, and η is the specified efficiency. Once the exit flow pressure and enthalpy are known a call to prop with enthalpy as the input determines the exit flow temperature and entropy.

The model's parameters are as follows.

- pres - exit flow pressure (20.0 atm). Input.
- eff - efficiency of the compression process (0.85). Input.
- power.work - the work required (watts) to accomplish the pumping action. Output. Like the compressor model, work consumed in the compression process will be indicated by a negative value of this parameter.
- fl - exit flow from the model. Output.

3.3.9. Diffuser (df) model

The diffuser model represents a gaseous flow diffuser. This model and the nozzle model are the only steady state models that make use of the flow velocity. The diffuser model has one calculational member function denoted c. The model requires one input flow from the gass stack and puts one output flow back onto the stack.

On entry to the model the total pressure p_t of the flow is determined by iterating on the pressure at constant inlet entropy until a value of the enthalpy equal to the total inlet enthalpy h_t is obtained, where

$$h_t = h + v^2 / 2$$

and h is the inlet enthalpy and v is the inlet velocity. Once this total pressure at the inlet is known, the exit values for the velocity, enthalpy and pressure are then determined from

$$v = v_{exit}$$

$$h = h_{in} - v^2 / 2$$

$$p = p_{in} + (p_t - p_{in}) / P_{rec}$$

where the subscript *in* corresponds to the inlet values and P_{rec} is the pressure recovery coefficient. Finally, a call to prop gives the exit values for the flow temperature, entropy, and density.

The model parameters are as follows.

vel - exit velocity (10.0 m/s) from the diffuser. Input.
 pres_rec - pressure recovery coefficient (0.5). Input.
 fl - exit flow from the model. Output.

3.3.10. Nozzle (nz) model

The nozzle model represents a gaseous flow nozzle. The model has one calculational member function c. It requires one input flow and generates one output flow.

The model makes use of a specified exit pressure p_{exit} and a call to the prop function with the inlet entropy value to determine the enthalpy h_s for an isentropic expansion to the exit pressure. The exit flow velocity is then determined by

$$v = \sqrt{v_{in}^2 + 2\eta(h_{in} - h_s)}$$

The exit flow enthalpy is then found from

$$h = h_{in} + (v_{in}^2 - v^2) / 2$$

and the rest of the exit flow's state variables are determined by a call to prop with the exit enthalpy as input. For use as output variables the exit Mach number, thrust and specific impulse are then calculated from

$$Mach = v \sqrt{(\partial \rho / \partial p)_s}$$

$$thrust = mv + pA$$

$$impulse = thrust / (9.8m)$$

where m is the mass flow rate, A is the exit flow area, and $(\partial \rho / \partial p)_s$ is calculated via finite differencing.

The input parameters are

pres - exit pressure (0.5 atm) of the nozzle.
 eff - efficiency of the nozzle (0.85).

The output parameters are

area - exit flow area (m^2) from the nozzle.
 mach - exit mach number from the nozzle.
 thrust - thrust (nt) generated by the nozzle.
 impulse - specific impulse (s) of the nozzle.
 fl - exit flow from the model.

3.3.11. Combustor (cb) model

The combustor model is used to burn a fuel with an oxidizing gas flow. The fuel is described by the input parameters of the model while the oxidizing flow is taken from the gas stack, and must be a flow with a "GAS" id. The model has one calculational member function denoted as c which takes one input flow from the stack and puts back one output flow.

On entry to the model a reference gas calculation is made to ultimately determine the heat of formation of the fuel. This is done by first calculating the mass flow rate of oxygen necessary to burn the fuel at a stoichiometry of one from

$$m_o = (2.6641 w_c + 7.93645 w_h + 0.99797 w_s - w_o) m_{fuel}$$

where w_c , w_h , w_s , and w_o are the weight fractions of carbon, hydrogen, sulfur, and oxygen in the fuel and m_{fuel} is the mass flow rate of the fuel. The molar flow rates for the carbon, hydrogen, sulfur, water, nitrogen, and oxygen (including m_o) for a reference gas oxidizing the fuel at stoichiometry of one are then determined from

$$mol_i = w_i m_{fuel} / mw_i$$

where the subscript i stands for one of the above species and the mw_i are the molecular weights for these species. By calling the prop function with for this reference combustion gas at a temperature of 298.16 K and a pressure of 1.0 atm a reference enthalpy as well as the equilibrium composition can be determined. In particular the amount of water vapor in the combustion products can be determined from the mole fraction of water in the gas composition. Knowing the higher heating value of the fuel HHV , the heat of formation of the fuel Δh_{form} can be determined from

$$\Delta h_{form} = (m_{fuel} + m_o)(h - h_{h_2o}) + m_{fuel} HHV / m_{fuel}$$

where h is the reference enthalpy calculated above and h_{h_2o} is the saturation vapor/liquid water enthalpy difference times the fraction of water within the combustion gasses given by

$$h_{h_2o} = 1050.65 * 2344.44 * 18.01534 * comp_{h_2o} / mw$$

where $comp_{h_2o}$ is the mole fraction of h_2o in the reference gas and mw is the molecular weight of the reference gas.

Once this reference gas calculation is done the actual oxidizing flow can be used to determine the actual stoichiometry of the combustion from

$$stoich = comp_{o_2} \frac{m_{ox}}{mw_{ox}} \frac{31.9988}{m_o}$$

where $comp_{o_2}$ is the mole fraction of o_2 in the oxidizing flow, m_{ox} is the mass of the oxidizing flow, and mw_{ox} is its molecular weight. The molar flow rates of the actual combustion gas species consisting of the original fuel species and the actual oxidizing flow species can be determined from a simple addition

$$mol_i = (comp_i)_{ox} \frac{m_{ox}}{mw_{ox}} + (mol_i)_{fuel}$$

Here the $(mol_i)_{fuel}$ here does not include the m_o as used in calculation of the reference gas. These molar rates can be normalized to yield the combustion gas species mole fractions which can then be used through a call to the atom function to determine the atom fractions for the combustion gas. The enthalpy and mass flow rate of this gas is then determined from

$$h = \frac{m_{ox} h_{ox} + m_{fuel} \Delta h_{form}}{m_{ox} + m_{fuel}}$$

$$m = m_{ox} + m_{fuel}$$

A call to the prop function with this enthalpy as the input (and at the pressure of the oxidizing flow) will then give the flame temperature of the combustion products as well as their equilibrium composition and other state variables.

For use in power summaries, the input power to the combustor is stored as

$$Pow = m_{fuel} HHV$$

The input parameters to the model are

- mass - the mass flow rate (1.0 kg/s) of the fuel.
- carb - the carbon weight fraction within the fuel (0.25).
- h - the hydrogen weight fraction within the fuel (0.75).
- o - the oxygen weight fraction within the fuel (0.0).
- s - the sulfur weight fraction within the fuel (0.0).

n - the nitrogen weight fraction within the fuel (0.0).

h2o - the water weight fraction within the fuel (0.0).

hhv - the higher heating value (J/kg) of the fuel (1e7).

The model will generate three additional output parameters:

stoich - ratio of oxygen within the oxidizer to the amount of oxygen just necessary for 100% fuel oxidation.

power.heat - total thermal power input equal to hhv time the fuel mass.

fl - combustion gas flow from the model.

The combustor model should only be used with oxidizing flows having the "GAS" id. In addition the gas properties codes should be compiled with, at least, the following species - C, CO, CO₂, H₂, H₂O, S, SO₂, and N₂.

3.3.12. Power (power) model

The power model is somewhat different than the other models in that it does not process a particular flow, but instead, makes use of a power class used by the other models. Each model that produces work, such as the gas turbine, or consumes work, such as the pump, will record this information in the work parameter of a power class. Similarly, models that input heat, such as the combustor, or lose heat from the system, such as a ht model with a negative heat load, will record this information in the heat parameter of this power class. Each of the model's power class is then put onto a stack, denoted as pows, by calling the put member function of this stack with the model's power class as an argument. The power model then makes use of this pows stack to calculate the net work and heat associated with the entire system. This is done by calling the c member function.

The model has no input parameters, but does calculate the following output parameters.

prod - sum of all the positive power.work variables of all models.

cons - sum of the absolute values of all negative power.work variables of all models.

input - sum of all positive power.heat variables of all models.

lose - sum of the absolute values of all negative power.loss variables of all models.

Note even if the power model is not called the pows stack is available for printing out tables of the models' power classes via pows.print. This power class gives an example of how one can utilize information from all of the other models and process it in a global way. Thus, one could add cost, reliability, or some other subclass to each of the model classes and then add a system cost, reliability, or some other model to produce global system parameters just like this power model.

3.4. Steady-State Example One

Consider a system consisting of a hydrogen tank, a compressor, a heater, and a gas turbine connected together in that order. Such a system could be analyzed using the following input to GPS.

```
cinit
[/gas /gas1 /id "THR-H2" /t 300 /p 1.0 /m 1.0] cnew
[/cp /cp1 /pres 6.0 /eff 0.88] cnew
[/ht /ht1 /temp 1000.] cnew
[/gt /gt1 /pres 1.0 /eff 0.84] cnew

gas1.c cp1.c ht1.c gt1.c

gass.print mods.print
/all cdc
```

Here we use class gas (gas flow initiator) to represent the hydrogen tank and define a specific instance of that class as gas1. Parameter values are then assigned to initialize the flow. These include defining the flow as a hydrogen gas using "THR H2", and then defining the values of temperature, pressure, and mass flow-rate using t, p, and m. Instances of the compressor, heater, and gas turbine classes are then defined with their associated

parameter values. Note that all input parameters have default values and thus, could be left out if the defaults are appropriate for the problem. Finally, the calculational functions for each class instance are called in the order necessary for the problem and the gas stack and model stack print functions are called to obtain the results. As can be seen the largest part of the coding is supplying the model parameter values, which is usually the case.

Before defining more complicated system configurations, the mechanism for handling the fluid flow (or for that matter any type of flow) between the models needs to be considered. In the original SALT code, these flows were defined as structures that were simply passed to the models as arguments. The user of the code basically indicated the flows as either pass-through, inputs, or outputs. This limited the models to a fixed number of flows and also required that the flows be declared and known by the driver coding. In this implementation a new way of handling the flows was developed that does not require them to be known within the driver.

Basically, each model will take off of a flow stack, *gas*, the number of input flows that it requires and put onto the stack the output flows that it generates. Actually, only the address of the flows are saved on this stack, but the concept is the same. Thus, in the previous example, the *gas1.c* model, being an initiator of a flow, simply put one output flow onto the stack, *cp1.c* then took this flow off the stack and on completion of its calculations, put its output flow back onto the stack. Models *ht1.c* and *gt1.c* then did exactly as the *cp1.c* model taking their single input flow off the stack and putting their single output flow back onto the stack.

In order to handle arbitrary system configurations, where some flows may not be used by the next model in the flow path, it is only necessary to be able to remove a flow from the stack and, at a later point, place that flow back on the stack for further processing. For the *gastype* flows, this is done by two additional member functions in the *gas* model class. These two functions are *sav* for saving the flow (i.e. remove from the stack) and *rec* for recovering the flow (i.e. put back on the stack).

As an example, suppose we have a system consisting of a flow initiator, *gas1.c*, a flow divider¹, *dv1.c*, and two heaters, *ht1.c* and *ht2.c*, one for each flow out of the divider. The system configuration could be represented by

```
gas1.c dv1.c gas2.sav ht1.c gas2.rec ht2.c
```

Here we have used the *sav* member function of a second *gas* model, *gas2.sav*, to save the second flow from the divider (i.e. last flow out of the divider is on the top of the stack). The *ht1* will then pick up the first flow from the *dv1* and process it. The recover entry of *gas2* will place the saved flow back on the stack to be processed by the second heater, *ht2*.

In reality, this example only shows the pattern of using the *save* and *recover* functions, since in this case, we have lost the flow from the *ht1* model for further processing unless a third *gas* model is used to save it before restoring the flow from *gas2.rec*. Flows that are generated by a model and not used immediately by a subsequent model before other flows are generated are lost. At any point where a model is called, one only needs to look at the previous models to determine what the input flows may be. If a model requires two input flows and the previous model only generates one output, then, the model before the previous will also be used to obtain an input flow. In this case the model providing the first input flow should probably be one not requiring an input flow itself, like the *gas.rec* function. It is the responsibility of the user to correctly sequence the models to represent the system configuration.

Since models that have multiple input or output flows would place these flows on the flow stacks in a particular order, it becomes necessary to remember this order to properly save the flows for later processing. Thus, most of the models that have such multiple inputs or outputs have secondary functions that do the saving and recovering within the model class itself. For example, if the flow splitter is used rather than the fictitious flow divider model in the above example, then the inputs would look like the following.

```
gas1.c spl.c ht1.c spl.s ht2.c
```

Here, the *spl* function only places one flow back onto the *gas* stack for processing by the *ht1* model. The *s* function of the splitter model will then place the second or split-off flow onto the flow stack for processing by *ht2*. Use of these secondary model functions often eliminate the use of the *gas* model's *save* and *recover* functions and also provide a clearer representation of the system configuration. They do not, however, completely

¹This flow divider is not actually in the current model library, but is only for illustration.

eliminate the use of the gas save and restore functions. For example, a gas.sav would still technically be needed after ht1 if the output of this model were to be further processed. These secondary functions generally perform no modeling, thus, for generating output flows, they should be called only after the primary model function is called and, for obtaining input flows, they should be called before the primary model function is called. An example of the latter is the use of the mixer model's mx.s function which saves the input flow for later use when the primary mx.c function is called.

In addition to the gastype flow class, other types of flows may exist. For example, the dynamic models to be discussed in the next chapter make use of a shfttype flow class, representing the power extracted or delivered to a model by a shaft. These other flows are passed between the models on a stack (a different stack for each flow type) exactly like the gas flows. Thus, a model may pick up a gas flow from the gas flow stack gass and a shaft flow from the shaft flow stack, which is denoted as shfts. The different flow stacks are entirely independent of each other. Thus, in determining the flow inputs to a model by considering the previous model what is really meant is the previous model generating an output flow of the correct flow type. For example, a shaft flow might be generated and then many models might be called requiring only gastype flows before the model that requires the shfttype flow is called.

3.5. Steady-State Example Two

The last example shows how to set up a very simple gas turbine system, however, that system is not too realistic. A better example might include some constraint on the power generated by the system to be fixed as some value, say 40 MW. This constraint which is dependent on more than one component of the system must be specified by the user. Constraints like this are system related as opposed to being component model specific, and, depending upon the system analysis being done, such constraints may not always be required. Thus, for generality, these system constraints are not automatically established by some builtin procedure. Additionally, these constraints may often be established in more than one way. For example, in this case, one might be able to establish this constraint by varying the pressure levels or by varying the mass flow rate. In any case the imposing of the constraint is not difficult and is nothing more than performing an equation solving task, where the constraint equation is the system power equal to 40e6 and the parameter to be varied is, say, the mass flow rate - gas1.m. Adding this task to the input of example one, the input becomes as follows.

```

cinit
[/gas /gas1 /t 300 /p 1.0 /m 1.0 /id "THR-tH2"] cnew
[/cp /cp1 /eff 0.88 /pres 6.0] cnew
[/ht /ht1 /temp 1000] cnew
[/gt /gt1 /eff 0.85 /pres 1.0] cnew
[/task /a] cnew
{a.c}
  {/gas1.m 1. 0.1 50.0 vary
    gas1.c cp1.c ht1.c gt1.c
    /gas1.m (cp1.power.work+gt1.power.work-40e6) cons
  }
while
gass print mods.print
/all cdel

```

Here, the task added was denoted as a, and the gas1.m (mass flow rate) variable was varied between 0.1 and 50.0 starting at 1. After the model calculational entries were called, the value of the power consumed by the compressor, which is denoted as cp1.power.work, and the value of the power generated by the gas turbine, gt1.power.work, will be known and the constraint can then be specified. When the while loop for this task a has converged, the constraint will be equal to zero (to the default error tolerance, since none was specified).

3.6. Steady-State Example Three

Continuing with the preceding example, one might desire a particular exit turbine temperature or some other constraint. These problems are solved by simply using additional vary and cons operators. Or one might want to optimize the efficiency subject to various constraints, both equality and inequality. These problems also are solved by simply using additional vary, cons, icons, and mini operators. Here we add a parameter sweep to the

previous problem. Suppose it is desired to look at the previous system for different heater exit temperatures of say, 800, 1000, 1200, and 1400. This is accomplished by simply putting a `for` loop around the task loop and the `gass` and `mods` print functions. The input in these case would be as follows.

```

cinit
[/gas /gas1 /t 300 /p 1.0 /m 1.0 /id "THR-tH2"] cnew
[/cp /cp1 /eff 0.88 /pres 6.0] cnew
[/ht /ht1 /temp 1000] cnew
[/gt /gt1 /eff 0.85 /pres 1.0] cnew
[/task /a] cnew
800 200 1400
  {/ht1.temp exch def
    {a.c}
    {/gas1.m 1. 0.1 50.0 vary
      gas1.c cp1.c ht1.c gt1.c
      /gas1.m (cp1.power.work+gt1.power.work-40e6) cons
    }
    while
      gass.print mods.print
    } for
  /all cdel

```

Here the starting value of 800, and increment of 200, and upper bound of 1400 are pushed onto the stack then the task loop and the output functions are inserted into a new procedure followed by the `for` operator. Since the `for` operator pushes onto the stack the value being iterated over, the first line of the new procedure takes this value and assigns it to the `ht1.temp`. Note that the print functions must be within the `for` loop otherwise only the results for the last value of the heater temperature would be printed. In this case the temperature values are equally spaced and a `for` loop could be used. Alternatively a `forall` loop could be used with the previous starting, increment, and upper bound values simply replaced by an arbitrary array of values, for example, [800 925 1130 1385]. Additional sweeps can be done simply by nesting other iterative loops around that for the heater temperature loop.

3.7. Steady-State Example Four

For the next example we consider a somewhat more realistic example, a diagram of which is shown in Figure 1². This system is of a simple space propulsive system. First, we consider the system formulated without any constraints.

In Figure 1 a gastype flow is initialized using an instance of the gas model, which has been named `gas_h2`. As a convention in naming the models we will use the model class type, an underscore, and a label. Although, the flow being initialized is a gastype, as explained previously, this really refers to the flow class type structure, and not that the flow needs to be a gas. In this case the `gas_h2` model parameters will be defined to initialize a hydrogen flow within the liquid region. This hydrogen flow is then passed through low pressure and high pressure pumps, denoted `pump_lp` and `pump_hp`. The flow then passes through a heat exchanger `hx_nz`, representing the nozzle cooling. Note, the current nozzle model does not include the provisions for a coolant flow, thus, this heat exchanger is used to simulate these effects. The flow is then split using a splitter, `sp_2`, into a main flow and a second flow which is further split using `sp_1`. These last two flows are then passed through two gas turbines, `gt_hp` and `gt_lp` which are used to drive the low and high pressure pumps. These gas turbine flows are then mixed together in `mx_1` and then mixed back into the main flow in `mx_2`. The resulting flow is then passed through a heater model used to simulate a reactor, denoted as `ht_reac` and then through the hot side of the `hx_nz` model and out the main thruster nozzle, `nz_1`.

In formulating the inputs we will start with the model calls necessary to describe the system configuration. This will be done exactly like the simpler examples described above by simply listing the models in the order that they process the gastype flows and using the secondary splitter and mixer functions where necessary. Note,

² Note that system diagrams such as shown in Figure 1 can be semi-automatically generated through the GPS input itself as will be discussed in a later chapter.

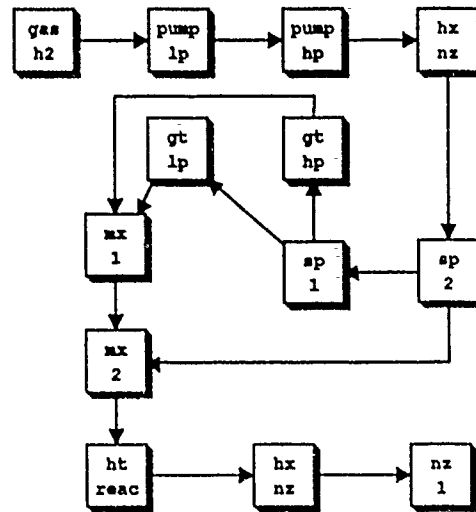


Figure 1.

depending on which flow from the splitters are treated as the primary flow and which are treated as the split-off or secondary flow, different system representations can be defined. Here we will assume that the primary flow from `sp_2` passes through `sp_1` and that the primary flow from `sp_1` passes through the `gt_lp` model. Thus, the system is described up to the secondary function of the `mx_1` model by

```
gas_h2.c pump_lp.c pump_hp.c hx_nz.c sp_2.c sp_1.c gt_lp.c mx_1.s
```

At this point, the secondary function of the `sp_2` model can be called to retrieve its split-off flow which is then processed into the secondary function of the `mx_2` model using

```
sp_2.s mx_2.s
```

Then the secondary function of the `sp_1` model can be called to retrieve its split-off flow. The rest of the models can then be called in the order that they process the flows as follows.

```
sp_1.s gt_hp.c mx_1.c mx_2.c ht_reac.c hx_nz.h nz_1.c
```

Note, that here the mixer models will use the flows previously saved by their secondary functions. Also, note that the hot side function is being called for the `hx_nz` model. The entire system configuration is thus represented by

```
gas_h2.c pump_lp.c pump_hp.c hx_nz.c sp_2.c sp_1.c gt_lp.c mx_1.s
sp_2.s mx_2.s sp_1.s gt_hp.c mx_1.c mx_2.c ht_reac.c hx_nz.h nz_1.c
```

For each of the models used within the system one will need to allocate an instance of the model and to define the appropriate model parameter values. These are, of course, completely dependent on the problem. For example, the `gas_h2` model instance and its parameter values might be defined as follows.

```
[/gas /gas_h2 /id "THR-tH2" /t 20 /p 1.29 /m 7.387 /v 200] cnew
```

Here we define the model with the name `gas_h2` and then assign its flow id parameter the value "THR-tH2". (Note, the "THR-tH2" represents a hydrogen gas flow, the small 't' before the H2 denotes a special version of the hydrogen data valid for high temperatures.) The rest of the line then represent the chosen initial values for the temperature, pressure, mass flow-rate, and velocity. The other models used would need similar allocations and are shown in the final inputs to the problem.

The rest of the inputs to this problem are formed by adding the `cinit` call and two additional function calls, one to print the gas flow output and one to print the model parameter output. We also include in this example a call to the power stack print function `pows.print`. Finally, we include a call to the `edcl` operator to delete the

model classes used by the problem. The final complete input for this problem is as follows.

```

cinit
[/gas /gas_h2 /id "THR-tH2" /t 20 /p 1.29 /m 7.387 /v 200 ] cnew
[/pump /pump_lp /eff 0.67 /pres 7.96 ] cnew
[/pump /pump_hp /eff 0.81 /pres 139.22 ] cnew
[/hx /hx_nz /t_cold (1043.0/1.8) ] cnew
[/sp /sp_2 /sr 0.3 ] cnew
[/sp /sp_1 /sr 0.3 ] cnew
[/gt /gt_lp /eff 0.23 /pres 85.91 ] cnew
[/gt /gt_hp /eff 0.75 /pres 85.74 ] cnew
[/mx /mx_1 ] cnew
[/mx /mx_2 ] cnew
[/ht /ht_reac /temp (5274/1.8) ] cnew
[/nz /nz_1 /eff 0.85 /pres 0.1 ] cnew
[/task /a] cnew

gas_h2.c pump_lp.c pump_hp.c hx_nz.c
sp_2.c sp_1.c gt_lp.c mx_1.s sp_2.s mx_2.s sp_1.s gt_hp.c
mx_1.c mx_2.c ht_reac.c hx_nz.h nz_1.c

gass.print mods.print pows.print
/all cdel

```

The outputs for this example, shown in Appendix B, are the results of the **gass.print**, **mods.print** and **pows.print** calls. The **gass.print** call displays the table of state points of exit flow from each model. All units in this table are in SI with the exception of pressure which is in atmospheres. Following the state point outputs are the individual model parameter outputs, which were generated by the **mods.print** call. Finally, the table of model powers - input, loss, produced, and consumed are generated by the **pows.print** function.

In looking at these outputs it can be seen that if the pair of models **gt_lp** and **pump_lp** were to form a turbo-pump then the power consumed by the pump should equal the power produced by the turbine which is not the case here. The same situation holds with the **gt_hp** and **pump_hp** model pairs. Thus, it would be more appropriate to constrain the power produced and power consumed in these two model pairs. This is nothing more than an equation solving task, similar to example two. The first step is to determine what parameters could be varied to establish these constraints. In general, there are usually many parameters that could be varied within a system in order to establish constraints. The only criteria is that the constraints be functionally dependent on the chosen parameters. In this problem the most obvious parameters would be the pressure levels at the exit of the models concerned. For this problem, however, the pressure levels out of the models represent the pressure leading to the reactor and the nozzle, and thus, are important parameters of the problem. It would be best to fix these at the appropriate design level and vary some other parameters. Another set of parameters might be the split ratios at the two splitters. By varying these split ratios varying amounts of mass flow can be directed to the two turbines generating more or less power. Using these split ratios the vary statements would then look like the following.

```

/sp_2.sr 0.3 0.1 0.9 vary
/sp_1.sr 0.3 0.1 0.9 vary

```

where here the starting values, lower and upper bounds were taken as 0.3, 0.1, and 0.9, respectively. The constraints can be taken as

```

/sp_2.sr (gt_lp.power.work+pump_lp.power.work) cons
/sp_1.sr (gt_hp.power.work+pump_hp.power.work) cons

```

Here the constraint delimiters (the first argument) have been taken as the two split ratios and the actual constraint expressions as the sum of the respective models power.work variables. Note, this variable is algebraic with negative values meaning work consumed and positive work produced. Thus, the sum is used rather than a difference to equate work consumed with work produced. Including the declaration of the task itself and adding

these vary and cons statements to a task loop around the model calls is all that needs to be done to establish these system constraints. The new complete inputs are as follows.

```

cinit
[/gas /gas_h2 /id "THR-tH2" /t 20 /p 1.29 /m 7.387 /v 200 ] cnew
[/pump /pump_lp /eff 0.67 /pres 7.96 ] cnew
[/pump /pump_hp /eff 0.81 /pres 139.22 ] cnew
[/hx /hx_nz /t_cold (1043.0/1.8) ] cnew
[/sp /sp_2 /sr 0.3 ] cnew
[/sp /sp_1 /sr 0.3 ] cnew
[/gt /gt_lp /eff 0.23 /pres 85.91 ] cnew
[/gt /gt_hp /eff 0.75 /pres 85.74 ] cnew
[/mx /mx_1 ] cnew
[/mx /mx_2 ] cnew
[/ht /ht_reac /temp (5274/1.8) ] cnew
[/nz /nz_1 /eff 0.85 /pres 0.1 ] cnew
[/task /a] cnew

{a.c}
{
  /sp_2.sr 0.3 0.1 0.9 vary
  /sp_1.sr 0.3 0.1 0.9 vary
  gas_h2.c pump_lp.c pump_hp.c hx_nz.c
  sp_2.c sp_1.c gt_lp.c mx_1.s sp_2.s mx_2.s sp_1.s gt_hp.c
  /sp_2.sr (gt_lp.power.work+pump_lp.power.work) cons
  /sp_1.sr (gt_hp.power.work+pump_hp.power.work) cons
}
while
mx_1.c mx_2.c ht_reac.c hx_nz.h nz_1.c

gass.print mods.print pows.print
/all cdel

```

Note that within these inputs those models that appeared after the gas turbines which don't affect the constraints were not included within the task loop. This is only a computational performance issue, as all the models could be included if desired. In fact, the gas_h2, pump_lp, pump_hp, and hx_nz models could be put before the loop as varying the split ratios will not affect any of their outputs. If that were done then the system configuration and task loop would then look like the following.

```

gas_h2.c pump_lp.c pump_hp.c hx_nz.c
{a.c}
{
  /sp_2.sr 0.3 0.1 0.9 vary
  /sp_1.sr 0.3 0.1 0.9 vary
  sp_2.c sp_1.c gt_lp.c mx_1.s sp_2.s mx_2.s sp_1.s gt_hp.c
  /sp_2.sr (gt_lp.power.work+pump_lp.power.work) cons
  /sp_1.sr (gt_hp.power.work+pump_hp.power.work) cons
}
while
mx_1.c mx_2.c ht_reac.c hx_nz.h nz_1.c

```

The resulting output for this example is shown in Appendix C. The only difference between this output and that within Appendix B is the inclusion of the task loop iterations and the resulting changes within the mass flow rates through the system.

CHAPTER 4

Dynamic Power System Model Classes

4.1. Introduction

This chapter discusses the components that are used to perform a dynamic power system analysis. These models consists of the following.

gas -	gas flow initiator
sp -	gas flow splitter
mx -	gas flow mixer
ht -	hgas flow heater/cooler
reac -	nuclear reactor
gt -	gas turbine (based on a performance map)
cp -	compressor (based on a performance map)
pump -	pump
exnz -	exhaust nozzle
pi -	pipe
valv -	valve
cntl -	PID controller
shft -	shaft flow initiator
mot -	motor
gen -	generator.

Note that the names of some of these models are exactly the same as those in the steady-state collection. Actually the steady-state collection could be combined with these dynamic ones, however, they have been kept separate for two reasons. The first is that the dynamic models often require much more input information than the steady-state models. Secondly, many of the dynamic models actually make use of the `diff` function in order to define the ordinary differential equations of the model and, in some models, also use `vary` and `cons` calls to set up the algebraic equations representing the fluid mass/momentum transfers. This means that the user must be more familiar with the use of these models and, in particular, will need to define the appropriate task loops within the inputs.

4.2. Dynamic Model Flow Classes

The present collection of dynamic models make use of two flow classes. The first is that of `gastype` and is exactly the same as that used by the steady-state model collection. The other flow type used, at present, is that of `shfttype`. This flow type is used to represent the information present in a shaft and consists of the following variables.

rpm -	shaft's rpm
inertia -	cumulative polar inertia of all components on the shaft
power -	cumulative net power delivered to the shaft

The `shfttype` flows are all placed onto a stack, denoted as `shfts`. The `shfttype` class has several member functions one denoted as `shftget` is used to retrieve a flow from this stack and like the `gasget` is only used internally by the models. The `shfts` stack also has a member function `print` which like the `gass.print` function can be used

to print out tables of the exit shaft flow from the component models.

Like the gas model, the `shfttype` class has a `shift` model that is used to initiate the `shfttype` flow and, to save and recover the flow from the `shfts` stack.

4.3. Dynamic Models

At present, the dynamic models require two different task loops. The first is the task `dyn` and represents the dynamic integrations over time and the second, is the task `sta` and represents the calculations of the mass / momentum transfers within the system. This second task loop should be nested within the dynamic task. A more detailed discussion of these matters will be presented after the individual models have been presented. Since both of these tasks must be defined for these dynamic models the allocations of these two tasks is carried out for the user within the `cinit` function, and thus, no `cnew` calls need to be made for these two tasks.

Some of the dynamic models use a modeling that is relative to rated or design point values. Thus, to properly use these models, the system under analysis must be set up and run at this design point first. This design point run will then yield values that are put back into the model as inputs for doing an off-design run, such as a system startup. In doing the design point run some of the task functions used by the models may need to be turned off. This is uniformly done by using the model's `stat` variable. Additionally, the dynamic integrations performed by the `dyn` task would not be done for this design point run and thus, do not need to be set up within the driver coding.

In addition to these design point options, many of the models have options for setting up the initial values at the off-design starting points. For example, one may wish to set a particular temperature distribution within some heat exchanger or start with some particular fuel temperature within a reactor. Since no special model functions are called to perform these initializations outside of the task loops that the user has coded within the inputs, the models need some way of determining that these initialization calculations are necessary. This is accomplished by using the state variable within the `dyn` task. As previously mentioned, state is used to control the integration procedure. In addition, it is used by the dynamic models to inform them when the initialization calculations are needed. When `dyn.state` is zero the integration over time has not started. Thus, the dynamic models can check this variable and perform their needed initializations or other calculations needing to be done before the integrations start. It should be noted, however, that when a `sta` task loop is nested within the `dyn` task loop the models will be called many times while `dyn.state` is zero. In the description of the models below the calls where `dyn.state` is zero are referred to as the initializing calls.

4.3.1. Gas (gas) model

As with the steady-state model, the gas model is used to initialize a `gastype` flow. The model has the initializing member function denoted as `c`. This function requires no input flows but does put one output flow onto the `gass` stack.

The modeling consists basically of assigning the specified inputs to the models exit flow values of `id`, temperature, pressure, mass flow rate, quality, and composition.

$$id=id_0$$

$$t=t_0$$

$$p=p_0$$

$$m=m_0$$

$$q=q_0$$

$$comp_i=comp_{0,i}$$

where the subscript 0 indicates user specified values. The model then calls the `atom` function to determine the atom fractions of the chemical species within the flow (note, that this is really only needed for "GAS" type flows). A call to the `prop` function with temperature as the input then furnishes the exit flow values of enthalpy, entropy, and density. The exit flow area, A , is either directly input or calculated based on an input diameter, d ,

assuming a circular flow cross section.

$$A = \pi d^2 / 4$$

The exit flow velocity is then calculated from

$$v = m / (A \rho)$$

where ρ is the fluid density at the exit.

The gas model's *c* function will also call *vary* to vary the flow's mass flow rate, *m*, if the input variable *stat* is set to zero. If *stat* is one then the mass flow rate is simply taken as *m* and is not varied. The constraint to go along with this *vary* will be defined within some downstream model.

The gas model also has a function to save a flow from the gas stack, denoted as *sav*, and a function to recover the flow, denoted as *rec* for representing complex system configurations.

The model's parameters are as follows.

<i>id</i> -	flow's id pointer ("THR-tH2"). Input.
<i>t</i> -	flow's temperature (298.16 K). Input.
<i>p</i> -	flow's pressure (1.0 atm). Input.
<i>m</i> -	flow's mass flow rate (1.0 kg/s). Input.
<i>v</i> -	flow's velocity (m/s). Output.
<i>comp</i> -	flow's species mole fraction. Input.
<i>area</i> -	flow's exit area (m^2). Input or output.
<i>diam</i> -	flow's exit diameter (0.1 m). Input or output.
<i>stat</i> -	specifies whether or not the steady-state option is in effect (0). Input. Stat equal to one turns the option on, zero turns it off.
<i>fl</i> -	exit flow from the model. Note that <i>fl</i> will need to be further qualified with the name of a particular gastype parameter, such as "fl.t".

The *id* pointer should be assigned a value of either "GAS" or "THR-species" as described within the discussion of the gastype flow class. At present, there is no options for starting a flow out in the two phase region as with the steady-state gas model. If *diam* is specified as zero then the *area* parameter is used to determine *diam*, otherwise *diam* is used to determine *area*. Thus, either one or the other of these variables should be input.

4.3.2. Shaft (shft) model

The *shft* model is used to initialize a *shfttype* flow using the *c* member function. This function requires no input flows and puts one output flow onto the *shfts* stack.

The modeling within the *shft* model's *c* function consists of initializing the *shfttype* flow.

$$rpm = rpm_0$$

$$inertia = 0$$

$$power = 0$$

where rpm_0 is the specified input value.

The shaft model also has a function to save a flow from the *shfts* stack, denoted as *sav* and a function to recover the flow, denoted as *rec*. These functions are used exactly as with the *save* and *recover* functions within the gas model for representing complex system configurations.

In addition, the shaft model has an *end* member function which is used to define the differential equation for the shaft speed. This function requires one input *shfttype* flow from the *shfts* stack and uses the information within this flow of the cumulative totals of *power*, *power*, supplied to the shaft and the cumulative totals of polar moments of inertia, *I*, of models on the shaft to define the time rate of change of the *rpm* as follows.

$$\left(\frac{2\pi}{60}\right)^2 \text{rpm} \mid \frac{d\text{rpm}}{dt} = \text{power}.$$

The model then calls the task related diff function with rpm as the variable to be integrated. Thus, this end function needs to be called in order to correctly model the speed-up or slow-down of a shaft. Since this function will treat the rpm variable as a state variable it should not be input as a function of time within the driver coding. If it is desired to change the rpm of the shaft directly as a function of time (to simulate some additional power input or output) then this end function should not be called.

The model's parameters are as follows.

rpm -	initial revolutions per minute of the the shaft (0.0). Input and output.
drpm -	time rate of change of rpm. Output.
inertia -	total moment of inertia of all the components on the shaft (kg m ²). Output.
power -	net power supplied to the shaft (w). Output.
shftf -	exit shftype flow from the model's c function. Output. Like the output gastype flows from other models, shftf will need to be further qualified with one of the shftype parameters, such as "shftf.rpm".

4.3.3. Compressor (cp) model

The compressor model is used to model a simple gastype flow compression process. The model is based on performance maps rather than physical modeling mainly to keep the model generic in nature. The performance maps are obtained by calling the in member function which will read the maps from a file. Thus, this in function needs to be called once before the calculational function is called. The calculational function is denoted as c. C requires one gastype input flow from the gass stack and puts one output flow back onto the stack. It also requires one shftype flow from the shfts stack and puts one shftype flow back onto this stack.

The modeling used in the compressor is dependent on whether the model is being called at a design point or an off-design point. If at the design point, the model calculates a rated corrected mass flow parameter $c_{mass, rated}$ and a rated corrected speed parameter $crpm_{rated}$ to be used in off-design runs.

$$c_{mass, rated} = m_{in} \sqrt{t_{in}} / p_{in}$$

$$crpm_{rated} = rpm_{in} / \sqrt{t_{in}}$$

where m_{in} is the inlet mass flow rate, p_{in} is the inlet pressure, t_{in} the inlet flow temperature, and rpm_{in} in the inlet shaft rpm. The rest of the modeling is the same whether at the design point or off-design. A corrected mass flow parameter c_{mass} and corrected speed parameter $crpm$ are calculated from

$$c_{mass} = \frac{m_{in} \sqrt{t_{in}} / p_{in}}{c_{mass, rated}}$$

$$crpm = \frac{rpm_{in} / \sqrt{t_{in}}}{crpm_{rated}}$$

Note that, for the design point these values simply become 1. The performance maps are then called with these two corrected parameter values to obtain a pressure ratio pr_{map} and an efficiency η_{map} . In order to make use of the same performance maps for different sized compressors, these returned map values are further scaled as follows.

$$pr = 1 + (pr_{map} - 1) \frac{(pr_{rated} - 1)}{(pr_{rated, map} - 1)}$$

$$\eta = \frac{\eta_{rated}}{\eta_{rated, map}} \eta_{map}$$

where the subscripts *map* refers to the quantity obtained from the map, *rated* refers to the input rated quantity, and *rated, map* refers to the quantity from the map at the rated conditions.

The pr value is then used to determine the exit pressure

$$p = pr \cdot p_{in}$$

which is followed by a call to the prop function with the inlet entropy as input to determine the enthalpy h_s of an isentropic compression to the exit pressure. The actual exit flow enthalpy h is then determined from

$$h = h_{in} + (h_s - h_{in}) / \eta$$

and another call to the prop function with enthalpy as the input will then determine the other exit flow state values.

The power required by the compressor is then calculated as

$$Pow = m (h_{in} - h)$$

which is then added (note the power is calculated as a negative number) to the shaft's power, along with the compressor's moment of inertia.

The input parameters to the model are as follows.

- rat_cmass - rated or design point value of the corrected mass flow parameter. This value is obtained by running the model at the design point with the stat parameter specified as one, in which case, this parameter becomes an output value.
- rat_cspeed - rated value of the corrected speed parameter. This value is obtained as with the rat_cmass parameter.
- rat_pr - rated value of the compressor pressure ratio (outlet to inlet) (5.0).
- rat_eff - rated value of the compressor efficiency (0.8).
- inertia - polar moment of inertia (5.0 kg m²) for the compressor.
- file - character array holding the name of the file containing the performance maps ("cp.dat").
- stat - flag for turning on (stat equal one) or off (stat equal zero) the steady-state design point option (0).

Note, since the performance maps provide the pressure ratio and efficiency for the compressor there really is not much run-time input to the model. However, the model does need to be supplied the performance maps and does need to be run at the design point first in order to obtain rat_cmass and rat_cspeed. The two performance maps, one for the pressure ratios and one for the efficiency, are both stored in the same input file defined by the parameter file. The details of the data layout within the performance map file is described in Appendix G. The output parameters from the model are as follows.

- power - power (watts) required by the compression process. Note, that like the steady-state models, power is treated algebraically with negative values representing power consumed by the model. Thus, in a normal compressive process this parameter will be negative.
- eff - efficiency.
- cmass - the corrected mass parameter.
- cspeed - the corrected speed parameter.
- fl - exit gas flow from the model.
- shftf - exit shaft flow from the model.

4.3.4. Exhaust nozzle (exnz) model

The exhaust nozzle model is used to model a nozzle where the back pressure is effectively zero. Thus, the flow will be choked at the smallest cross sectional area. The model has options for handling both converging nozzles or converging/diverging nozzles. The input flow should be subsonic for the model to work properly. The calculational member function for the class is denoted as c. c requires one gastype input flow from the gass stack and puts one gastype output flow back onto this stack. Note that although a flow is put back onto the stack, this flow is only for printout and should not be used as input for any other downstream model.

The model starts by calculating the area, A , or diameter, d , of the throat section given one or the other of these variables and assuming a circular flow cross section. A check is then made to make sure that this flow area is less than the inlet flow area. If it is not the model terminates the run with a message. The model then determines the specific heat at constant pressure c_p at the inlet temperature of the fluid by calls to the prop function and using finite differencing. The gas constant R and ratio of specific heats γ for this flow are then determined assuming that the fluid is approximately an ideal gas.

$$R = p / (\rho t)$$

$$\gamma = c_p / (c_p - R).$$

These are then used to determine the inlet Mach number,

$$M = \frac{v}{\sqrt{\gamma R t}},$$

The stagnation temperature,

$$t_0 = t \left(1 + \frac{\gamma - 1}{2} M^2 \right)$$

stagnation pressure,

$$p_0 = p \left(\frac{t_0}{t} \right)^{\frac{\gamma}{\gamma - 1}}$$

flow temperature at the choke point or throat section,

$$t = \frac{2t_0}{\gamma + 1}$$

pressure at the choke point,

$$p = p_0 \left(\frac{t}{t_0} \right)^{\frac{\gamma}{\gamma - 1}}$$

and finally, the velocity at the choke point,

$$v = \sqrt{\gamma R t}.$$

Note, that usually, the models within GPS do not make use of such constant specific heat equations, preferring instead, to iterate over the property procedures. However, for the exhaust nozzle the above equations are reasonably accurate and eliminate the iterations over the prop function resulting in a faster running and more robust model.

Knowing the flow conditions at the throat and the area of the throat, the mass flow rate required at the throat is determined from

$$m_{req} = A p \sqrt{\frac{\gamma}{R t}}.$$

This mass flow rate defines a constraint on the inlet flow rate and, for the non-steady-state option is then used in a cons function call. For the steady-state option the inlet mass flow rate is assumed to be the correct value and the area at the throat is then calculated from

$$A = m \sqrt{R t} \sqrt{\gamma} / p$$

If a value was furnished to the variable A_{exp} , representing any additional expansion beyond the throat section, the flow is expanded by iterating over the exit Mach number, and using relations similar to above, calculating the exit flow temperature, pressure, and velocity, and hence, area. When the calculated area is the same as A_{exp} , the iterations are terminated, yielding the final exit Mach number as well as exit flow state variables.

For use in print out the thrust and specific impulse are calculated as follows.

$$thrust = mv + pA_{exp}$$

$$impulse = thrust / (9.8m).$$

Note, that when A_{exp} is not furnished the thrust calculation uses the throat area instead.

The parameters to the model are as follows.

diam -	throat (smallest) section diameter of the nozzle (0.05 m). Input.
area -	throat area (m^2). If diam is set to zero then area determines diam, otherwise diam is used to determine area.
acxp -	divergent section exit area ($0.0 m^2$). If acxp is less than area no divergent section is calculated.
mach -	initial estimate of the exit Mach number, only used when acxp is non-zero (1.5).
stat -	steady-state options flag, one for steady-state option on and zero for off (0). Input.
mreq -	mass required by the throat section. Output.
thrust -	thrust generated by the exiting flow (nt). Output.
impulse -	specific impulse of the nozzle (s). Output.
fl -	exit flow from the model. Output.

4.3.5. Gas turbine (gt) model

The gas turbine model is used to model a simple gastype flow expansion process. The model is based on performance maps rather than physical modeling mainly to keep the model generic in nature. The performance maps are obtained by calling the in member function which will read the maps from a file. Thus, this in function needs to be called once before the calculational function is called. The calculational function is denoted as c. C requires one gastype input flow from the gass stack and puts one output flow back onto the stack. It also requires one shfttype flow from the shfts stack and puts one shfttype flow back onto this stack.

The gas turbine model is very similar to the compressor model the only difference being the calculation of the exit flow enthalpy, which for the turbine is given by

$$h = h_{in} - \eta (h_s - h_{in})$$

and the calculation of the exit pressure

$$p = p_{in} / pr$$

where the notation is the same as that used in the compressor model.

The parameters to the model are as follows.

rat_cmass -	rated or design point value of the corrected mass flow parameter. This value is obtained by running the model at the design point with the stat parameter specified as one, in which case, the parameter becomes an output value.
rat_cspeed -	rated value of the corrected speed parameter. This value is obtained as with the rat_cmass parameter.
cmass -	the corrected mass parameter. Output.
cspeed -	the corrected speed parameter. Output.
rat_pr -	rated value of the expansion pressure ratio (inlet to outlet) (5.0). Input.
rat_eff -	rated value of the gas turbine efficiency (0.82). Input.
inertia -	polar moment of inertia ($kg m^2$) for the device. Input.
file -	character array holding the name of the file containing the performance maps. Input.
stat -	steady-state options flag, one for steady-state option on and zero for off (0). Input.
eff -	efficiency. Output.

- power - power (watts) generated by the expansion process. Output.
 fl - the exit gas flow from the model. Output.
 shfl - the exit shaft flow from the model. Output.

As with the compressor model, there are two performance maps for the turbine both stored in the same input file. The first supplies a pressure ratio as a function of the corrected mass and corrected speed and the second gives the efficiency as a function of the corrected mass and corrected speed. The details of the data layout within the performance map file is described in Appendix G.

4.3.6. Heater (ht) model

The heater model represents the transfer of heat into or out of a gastype flow. The model has one calculational member function denoted *c*. This function requires one input gastype flow from the gas stack and outputs one gastype flow back to the stack.

There are several options to this model. The first is a steady-state mode in which the heat input Q is simply added to the inlet flow enthalpy h_{in} to generate the exit enthalpy.

$$h = h_{in} + Q / m$$

where m is the mass flow rate of the fluid. The pressure drop Δp through the heater is given as

$$\Delta p = f_{rated} p_{in} \left(\frac{m}{m_{rated}} \right)^2$$

where the subscript *rated* refers to some input rated value and f is the fraction of the inlet flow representing the pressure drop. The exit flow pressure is then given by

$$p = p_{in} - \Delta p$$

and the other exit flow parameters are then obtained from a call to the prop function.

The second option is a simple thermal delay mode. This option permits a number of nodes along the device, with the exit enthalpies for each node given by

$$\frac{\partial h_i}{\partial t} = \frac{m (h_{i-1} - h_i) + Q_i}{K}, \quad i=1, \dots, n$$

where K is some input constant, n is the number of nodes, and Q_i is the amount of heat transferred at each node. Presently Q_i is taken as Q/n . In this mode the pressure at the exit of each node is defined as

$$p_i = p_{in} - \Delta p / n$$

where Δp is calculated as above. Note that this option is provided for those cases when one simply does not have enough information concerning the heater to make use of the full calculational mode but still desires some thermal delay to be included.

The final or full calculational mode again makes use of multiple nodes, with both a wall temperature and fluid enthalpies at each node calculated from the following.

$$c_p M_i \frac{\partial T_i}{\partial t} = Q_i - ua \Delta t_i$$

$$\rho V_i \frac{\partial h_i}{\partial t} = m (h_{i-1} - h_i) + ua \Delta t_i$$

where

$$\Delta t_i = T_i - (t_i + t_{i-1}) / 2.$$

Here T_i is the wall temperature at the i -th node, u is the overall heat transfer coefficient, a is the heat transfer surface area, c_p is the wall specific heat, and M_i is the wall mass at the i -th node. The exit pressure at each node is determined as with the previous mode and the other flow state variables are then determined with a call to the prop function with enthalpy as the input. The u that is used in the above equation is adjusted for changes

in mass flow rate from the rated flow using

$$u = u_{rated} \left(\frac{m}{m_{rated}} \right)^{0.8}$$

where u_{rated} is the heat transfer coefficient at m_{rated} . The flow properties at the exit of the heater are taken as those at the last node. Finally, for all modes, the exit velocity is calculated as

$$v = m / (\rho_i A)$$

where A is the exit flow area.

The input parameters to the model are as follows.

- num - number of internal nodes along the flow path within the model (0). If greater than zero, the total heat transferred is equally divided into this many parts with each part transferred per node. If equal to zero, the model works like a steady-state model with the resulting flow enthalpy changing instantaneously with the heat transfer.
- t#i - the flow temperature of the i-th node. This array, which should be defined for i from one to num, represents the initial gas temperature distribution. Thereafter t#i will be an output. If num is zero, this array is not used.
- heat - the total heat load on the exchanger (1e5 w). Generally, heat is used as an input only after the initial call to the model, however, heat can be used on the initializing call to generate the values of the t array.
- diam - diameter of the total flow passage through the device (0.1).
- area - area of the total flow passage through the device. If diam is zero then area is used to determine diam, otherwise area is determined by diam.
- length - length of the flow passage through the device (1.0 m). Note length and area are used to determine the volume of the device and not the heat transfer surface area which is supplied by ua.
- ua - heat transfer surface area time the heat transfer coefficient per node at the rated flow (1e5 w/K).
- rat_m - rated or design point mass flow rate through the device (5.0 kg/s). This is used to adjust the ua value and pressure drops for different off-design flow rates.
- rat_pf - rated or design point pressure drop through the device as a fraction of the inlet pressure (0.05).
- twall#i - wall temperature array. This array is used to give initial values for the gas passage wall temperatures. After the initializing call this array will be an output. Note that this array is used only if the wall is given a mass using mwall.
- mwall - mass of the walls containing the gas. If mwall is non-zero then the code will make use of the full calculational mode and calculate the wall temperatures. In this case the heat load is distributed uniformly along the wall.
- cpwall - specific heat of the wall material (1000 J/kg K).
- tconst - constant used for heat transferred to the gas in the simple thermal delay option (10). Tconst represents the K defined above.

As noted some of these variables are used as inputs only on the initializing call to the model and are subsequently defined as outputs. Thus, one needs to be cautious when calling this model within a loop. Iterating over different values of heat as inputs to establish some initial constraint, for example, will not work. This is because once the model is called with heat as an input, the t#i array is then defined, and hence on subsequent iterations, this t#i array would be used rather than the heat parameter. However, one could iterate over the t#i values. Output parameters from the model include

- heat0 - initial heat load on the device (w). This can be used on subsequent calls to give continuity to the thermal input using the heat parameter when the t#i array is used on the initializing call.
- vol - flow area times the heater length (m^3).
- fl - exit flow from the model.

4.3.7. Mixer (mx) model

The mixer model mixes together two gastype flows. The calculational member function is *c* and requires one input flow from the gass stack and puts one output flow back onto the stack. The other input flow is obtained by calling the member function *s*. This *s* function, which must be called before the *c* function, requires one input gastype flow but generates no output flow.

As with the most of the other dynamic models the exit flow area is calculated using a specified diameter *d* by

$$A = \pi d^2 / 4$$

or if the area is input the diameter is determined from the same equation. The exit mass flow rate is calculated from

$$m = m_1 + m_2$$

where the subscripts refer to the two inlet flows. The exit flow enthalpy is obtained from

$$h = \frac{m_1 h_1 + m_2 h_2}{m_1 + m_2}$$

and the exit flow velocity from

$$v = m / (\rho A)$$

where the exit flow density is obtained from a call to the prop function with enthalpy as the input.

In the dynamic mode the mixer model imposes a constraint on the entering flows that their pressures should be equal. This is done by calling the task class *cons* function. In the steady-state mode this constraint is not imposed. In either case, the exit pressure is defined to be the mass weighted average of the inlet pressures

$$p = \frac{m_1 p_1 + m_2 p_2}{m_1 + m_2}.$$

In a dynamic problem the inlet flow rates entering a mixer should rapidly adjust themselves such that the inlet pressures become equal. For the instantaneous representation of the mass / momentum effects used by the models these pressures are thus, simply constrained to be equal. The actual parameter that is varied to order to establish this constraint will appear within some other upstream model.

The parameters for the model are:

- diam - exit flow diameter (0.01 m). Input.
- area - exit flow area. If diam is zero, diam is determined by area, otherwise area is determined by diam.
- stat - flag used to turn on (stat=1) or turn off (stat=0) the dynamic mode constraint on inlet flow pressures (0).
- dp - difference in input flow pressures (atm). Output.
- fl - exit flow from the model. Output.

4.3.8. Pipe (pi) model

The pipe model models the pressure drop and optionally the enthalpy delays that occur within pipes. The model has a calculational member function, *c*, requiring one input gastype flow from the gass stack and puts one output flow back on the stack.

The pipe model is similar to the heater model in the way that it calculates pressure drops. The model also can make use of multiple nodes in representing enthalpy delays. Thus, the exit pressure and enthalpy from each node is calculated from

$$p_i = p_{i-1} - \Delta p / n$$

$$\bar{\rho}_i V_i \frac{\partial h_i}{\partial t} = m (h_{i-1} - h_i)$$

where $\bar{\rho}_i = (\rho_i + \rho_{i-1})/2$, V_i is the i -th node volume, and n is the total number of nodes. Finally, the exit flow velocity is determined from

$$v = m / (\rho A n_{par})$$

where A is the pipe, n_{par} is the number of pipes passing the flow in parallel. If the number of nodes is taken as zero, the model only calculates the pressure drop and no enthalpy delays are calculated.

The parameters for the model are as follows.

diam -	exit flow diameter (0.1 m). Input.
area -	exit flow area. As with the other models, if diam is non-zero, diam determines area, otherwise area determines diam.
length -	length of the pipe (1.0 m). Input.
rat_pf -	rated or design flow rate pressure drop as a fraction of the inlet pressure (0.01). Input.
rat_flow -	rated or design flow rate (1.0 kg/s). Input.
num -	number of nodes along the pipe length (0). Input. This is used to define the differential equations to simulate thermal delays. If num is zero, no thermal delays are modeled. For gas flows in short pipes num equal zero is generally sufficient.
num_par -	number of parallel flow segments or pipes in parallel that the model represents (1). Input.
dp -	total pressure drop along the pipe (atm). Output.
fl -	exit flow from the pipe. Output.

4.3.9. Pump (pump) model

The pump model, like the gas turbine and compressor, is based on performance maps rather than basic physical modeling. Again, this was done in order to have a generic model rather than a pump of a specific type. The performance maps are obtained using the member function `in`. This function should be called once before the calculational function is called. The layout of the data within the performance map file is discussed in Appendix G.

The calculational function is denoted `c`. This function requires one input gastype flow from the gass stack and puts one output gastype flow back onto this stack. The function also requires one shfttype flow from the shfts stack and puts one shfttype flow back onto the shfts stack. The pump model makes use of homologous pump head and torque curves. These take the head as a function of the normalized flow and rpm as follows.

$$m_n = \frac{m}{m_{rated}}$$

$$rpm_n = \frac{rpm}{rpm_{rated}}$$

$$x = \pi + \tan^{-1} \left(\frac{m_n}{rpm_n} \right)$$

$$H = (m_n^2 + rpm_n^2) H(x) H_{rated}$$

where H is the pump head, H is the pump head function, and the subscripts *rated* stands for rated values and n for normalized values. The torque T is calculated given the pump efficiency, η , as

$$T = \frac{60}{2\pi} \frac{m H}{\rho \eta rpm}$$

The torque can optionally be obtained from the homologous pump curves as

$$T = (m_n^2 + rpm_n^2) T(x) T_{rated}$$

and then the efficiency is calculated from the previous equation. (Note, that due to lack of good pump torque curves the first approach with the efficiency given is currently hardwired in the coding.) The exit flow state variables are then determined from

$$p = p_{in} + H$$

$$h = h_{in} + H / (\rho \eta)$$

and a call to the prop function.

The power required by the pump is calculated as

$$Pow = m (h_{in} - h)$$

which is then added to the exit shaft flow. In addition, the pumps moment of inertia is also added to the exit shaft flow.

The parameters for the model are as follows.

rat_m -	rated or design point mass flow (1.0 kg/s) for the pump. Input.
rat_rpm -	rated design point rpm (1000). Input.
rat_dp -	rated design point pressure rise (5.0 atm). Input.
rat_eff -	rated design point efficiency (0.85). Input.
eff -	pump efficiency (0.65). Input.
inertia -	polar moment of inertia (0.1 kg m ²). Input.
dp -	pressure rise across the pump (atm). Output.
power -	power required by the pump (w). Output. Like the compressor model, this parameter will appear as a negative number for power consumed.
fl -	exit gas flow from the pump. Output.
shaftf -	exit shaft flow from the pump. Output.
file -	file containing the performance maps ("pump.dat"). Input.

The model makes use of two performance maps both contained in the file defined by the parameter file. The first gives a nondimensional head curve as a function of the nondimensional flow rate and nondimensional rpm. The nondimensionalizing factor in all cases is the rated condition of the corresponding variable. The second performance map is of the nondimensional torque as a function of the same two nondimensional parameters. Note that unlike the gas turbine and compressor models, this model does not need to be run at the design point first.

4.3.10. Reactor (reac) model

The reactor model is also very generic in nature being predominately a heat exchanger, although, point kinetic equations are also provided as an option. The model has a main calculational member function denoted c. The function requires one input gastype flow and generates one gastype output flow.

If the point kinetics option is specified, the following equations are solved for the neutronics.

$$\frac{dc_i}{dt} = \frac{\beta_i P_f}{l^*} - \lambda_i c_i, \quad i=1 \dots 6$$

$$\frac{dP_f}{dt} = \frac{\bar{\rho} - \beta}{l^*} P_f + \sum_{i=1}^6 \lambda_i c_i$$

where P_f is the fission power, c_i is the precursor nuclei concentrations, λ_i is the radioactive decay constants, β_i is the i -th fraction of delayed neutrons, β is $\sum \beta_i$, $\bar{\rho}$ is the reactivity, and l^* is the prompt neutron generation time. The reactivity is presently taken as follows.

$$\bar{\rho} = \bar{\rho}_{cnl} + \bar{\rho}_{cool}(T_{cool} - T_{cool,0})$$

where $\bar{\rho}_{cnl}$ is the control reactivity, $\bar{\rho}_{cool}$ is the coolant feedback reactivity, T_{cool} is the average coolant temperature, and $T_{cool,0}$ is the initial T_{cool} . If the point kinetics option is not on, then the fission power is assumed to be an input.

The fission power is assumed to be dissipated in the fuel mass. The resulting fuel temperature, cladding temperature, and coolant flow exit enthalpy is then obtained from the following equations.

$$\frac{\partial T_{fuel}}{\partial t} = \frac{P_f - u_{clad}(T_{fuel} - T_{clad})}{cp_{fuel}M_{fuel}}$$

$$\frac{\partial T_{clad}}{\partial t} = \frac{u_{clad}(T_{fuel} - T_{clad}) - u_{cool}(\Delta T_m)}{cp_{clad}M_{clad}}$$

$$\frac{\partial h}{\partial t} = \frac{u_{cool}(\Delta T_m) + m(h_{in} - h)}{\rho vol}$$

where T is the average temperature, ΔT_m is the log mean temperature difference, h is the exit coolant enthalpy, h_{in} is the entrance coolant enthalpy, cp is the specific heat, M is the mass, ρ is the average coolant density, vol is the coolant passage volume, u is the heat transfer coefficient, and the subscripts denote whether the quantity is the fuel, cladding, or coolant fluid. The u_{cool} coefficient is also adjusted based on the fluid flow rate as follows.

$$u_{cool} = u_{cool, rated} \left(\frac{m}{m_{rated}} \right)^{0.8}$$

where $u_{cool, rated}$ is the heat transfer coefficient at the rated mass flow rate.

The pressure drop through the reactor is given exactly like the heater model based on a pressure drop fraction and adjusted as a function of the mass flow rate. The rest of the exit flow state variables are then determined from a call to the prop function with enthalpy as an input and the exit flow velocity is determined from

$$v = m / (\rho A)$$

As an option, the cladding mass may be set to zero in which case the calculation of the cladding temperature is not done.

The input parameters to the model are as follows.

- rat_m - rated or design point mass flow through the reactor (5.52 kg/s). This parameter is used to scale off-design pressure drops.
- rat_pf - rated pressure drop through the reactor as a fraction of the inlet pressure (0.2).
- tcool - initial value of the exit gas flow temperature (K).
- power - power level generated by the reactor (1e6 w). This is either an input for all time or, if the point kinetic equations are used, an initial value parameter.
- ucool - heat transfer film coefficient for the gas/cladding or gas/fuel interfaces at the rated flow (1.3e6 w/K).
- uclad - heat transfer coefficient between the fuel and cladding (1.3e6 w/K). Uclad is only used when mcald is non-zero.
- mfuel - mass of the fuel material (220 kg).
- mclad - mass of the cladding material (0.0 kg). If mclad is set to zero, no cladding material is assumed.
- cpfuel - specific heat of the fuel (1000 J/(kg K)).
- cpclad - specific heat of the cladding (1000 J/(kg K)).
- tclad - initial value of the cladding temperature. If tclad is set to zero, then it is taken as an equilibrium value (i.e. its time rate of change is zero) based on the power and heat transfer

- coefficients. Tclad is only used if mclad is non-zero.
- tfuel - initial value of the fuel temperature. If tfuel is set to zero, then an equilibrium fuel temperature based on the power and heat transfer coefficients is used.
- diam - diameter of the exit flow area (0.2 m).
- area - area of the exit flow. Diam and area are used to determine each other as with the other models.
- vol - volume of the gas passage within the reactor (1.0 M^3).
- reactcntl - control reactivity (0.0).
- reactcool - temperature coefficient of the reactivity (0.0). The total reactivity is taken as reactcntl plus this coefficient times the difference of the average coolant temperature and the initial average coolant temperature. The average is taken as the mean of the inlet and exit temperatures.
- option - flag used to turn on (option equal one) or off (option equal zero) the point kinetic calculations (0).
- beta#i - fraction of neutrons in the i-th delayed group, i goes from 0 to 5 (0.00021, 0.00141, 0.00127, 0.00255, 0.00074, 0.00027).
- bet - fraction of neutrons which are delayed (0.00645). Note, the code internally normalizes the beta array so that its elements sum to bet.
- lamb#i - radioactive decay constant of the i-th group of precursors, i goes from 0 to 5 (0.0124, 0.0305, 0.111, 0.301, 1.1, 3.0).
- The beta, bet, and lamb parameters are only used if option is set to one. The outputs from the model include
- tcool0 - initial average coolant temperature (K).
- power0 - initial power level (w).
- fl - exit flow from the reactor.

4.3.11. Splitter (sp) model

The splitter models the splitting of a gastype flow into two flows. The model has a calculational member function c which requires one input gastype flow and generates one output gastype flow. The other output flow is obtained by calling the member function s, which should be called only after the c function has been called.

The dynamic splitter model works much like the steady-state splitter model but only includes the full flow splitting and not the species splitting. Thus, given a split ratio sr the two exit flows, denoted by subscripts 1 and 2, are obtained from the following.

$$m_1 = sr \cdot m_{in}$$

$$m_2 = m_{in} - m_1$$

$$v_1 = m_1 / (\rho A_1)$$

$$v_2 = m_2 / (\rho A_2)$$

The temperature, pressure, enthalpy, etc. for the exit flows are the same as the input values. Note that the split ratio is varied for dynamic runs.

The parameters to the model are:

- diam#i - diameter (m) of the i-th exit flow with i being either 0 or 1 (0.01 m). Input.
- area#i - area of the i-th exit flow. If diam#i is zero the diam#i is determined from area#i, else area#i is determined from diam#i.
- stat - flag used to turn on (stat equal one) or off (stat equal zero) the steady-state option (0). Input.
- sr - split ratio of the second output flow to the inlet flow (0.5). Input. If stat is zero then this parameter only represents an initial value and is thereafter determined by some system

constraint.

fl# - exit flows from the model. Output.

The splitter model generates a new gastype flow and as such will vary the mass flow rate for the flow provided that stat is zero. This is done by varying the sr parameter. When stat is one, the specified split ratio is not changed.

4.3.12. Valve (valv) model

The valve model has two options, one using a conductance factor and valve position along with an empirical expression for the flow rate that can be passed as a function of the pressure drop, and a second option in which the pressure drop is directly input, the valve position being determined after the fact. The model has a calculation member function c requiring one input gastype flow and generating one output gastype flow.

In the first option, with the valve position *pos* and conductance factor *cv* specified, the fraction of inlet pressure *pf* representing the pressure drop across the valve is found by solving the equation

$$m = cv \text{ pos } (1 - pf / 3.0) \sqrt{\rho p_{in} pf}$$

where *m* is the inlet mass flow rate, *p_{in}* is the inlet pressure, and *p* is the inlet density. In the second option *pf* is directly input and the above equation is solved for the valve position assuming a conductance factor of one. In either case the exit pressure is then determined from

$$p = p_{in} - pf \text{ } p_{in}$$

and the other state values are determined with a call to prop with the inlet enthalpy as input.

The parameters to the model are

pf - ratio of the pressure drop to the inlet pressure (0.01). Input.
 option - flag specifying direct input of pf (option equal to two) or input of valve position and conductance (option equal to one) (2). Input.
 cv - valve conductance value. Input.
 pos - valve position, treated as a number between zero and one (1.0). Input.
 dp - pressure drop through the valve (atm). Output.
 fl - exit flow from the valve. Output.

If pf is input directly, then pos is an output actually representing the product of a valve conductance and a valve position.

4.3.13. Motor (mot) model

The motor models an additional supply of power to a shaft flow. The model requires one shfttype flow from the shfts stack as input and puts one shfttype flow back onto that stack on output.

The model has several different options, as specified by the parameter, option. If option is set to "zero", the motor will add to the input shaft flow power exactly the correct amount necessary to produce a zero shaft power at this point within the shaft flow. If the option is specified as "level", the model will add power to the shaft flow only if the current shaft flow power is less than some specified power level, *Power₀*. In this case, the power added will be the minimum of *Power₀* or the amount of power necessary to give the shaft flow a power level of *Power₀*. In other words, the motor will attempt to level the shaft power to *Power₀* at this point in the shaft flow path. Finally if the option is set to something other than "zero" or "level", the motor will simply supply to the shaft flow the specified power.

The parameters of the model are as follows.

option - character string representing the option as discussed above (""). Input.
 inertia - the moment of inertia for the motor (0.1 kg m²). Input.
 power - the specified input power level if option is not specified as "zero" or "level" (w). Input when option is not "zero" or "level", output otherwise.

power0 - specified $Power_0$ value as discussed above ($1e4$ w). Input when option has been set to "level".
 shftf - exit shaft flow from the model. Output.

4.3.14. Generator (gen) model

The generator extracts power from a shaft flow in an attempt to model an electrical generator. This model is only a very crude representation of such a generator. The model has the calculational function *c* which requires one input shfttype flow from the shfts stack and puts one such flow back onto that stack.

The model has two options controlled using the stat parameter. When stat is zero, the power required and extracted from the shaft is calculated as

$$Pow = -Pow_{rated} \left(\frac{rpm}{rpm_{rated}} \right)^4$$

where Pow_{rated} is some design point rated power level and rpm_{rated} is the rpm at this rated power level. When stat is one, the power required is assumed equal to the shaft power at this point in the flow path.

The model parameters are as follows.

inertia - polar moment of inertia for the generator (0.2 kg m^2). Input.
 rat_rpm - rated rpm (17800 rpm). Input.
 rat_pow - rated power ($40e6$ w). Input.
 power - required power (w). Output. Negative quantities represent power consumed.
 stat - design point or off-design point flag as discussed above (0). Input.

4.3.15. Controller (cntl) model

The controller models a proportional-integral-derivative (PID) controller. This model requires no input flows and generates no output flows. The controlling variable *var* is determined by the following equation.

$$var = var_0 + k \left(t_p \epsilon + \frac{1}{i_i} \int_0^t \epsilon + t_d \frac{d\epsilon}{dt} \right)$$

where ϵ represents the error between the value of the variable to be controlled and some specified desired value, var_0 represents a set point value of *var* at zero error, and k , t_p , t_i , and t_d represent adjustable parameters of the controller. Note, if t_i is specified as 0, then the integral term is not include. Likewise if t_d is specified as 0, then the derivative term will be zero. Thus, the controller also handles proportional, proportional-integral, integral, integral-derivative, etc. type of controllers. However, to actually, make use of the derivative type of controllers, one must be able to accurately calculate the derivative of the error. For GPS usage, this usually means only those variables that are state values of differential equations for which the derivatives are known can be used in derivative controllers.

Since the cntl model changes the controlling variable, the cntl model could cause problems in iterative sta task loop for dynamic problems. The controlling variable's changing value means that the constraints in this loop might not be a true function of the specified parameters that are being varied within the loop. To eliminate this problem, the cntl model has an option, controlled with the stat parameter, for putting the controlling parameter within a vary-cons function pair internally within the model. Thus, when stat is zero, a temporary value representing the value that the controlling variable is to obtain based on the above equation is calculated. *var* is then varied within the same sta task loop as the other parameters until it becomes equal to this temporary value. This permits the following method for using the cntl model. Within the inner loop of the dynamic problem, the actually controlling variable is defined to be equal to *var* before the model in which the controlling variable is found. After the error is calculated, the cntl model can then be called.

The parameters to the model are as follows.

stat - flag, when zero, turns on the implicit calculations of *var* using the vary-cons functions, and when one turns off these implicit calculations (0). Input. Note that when stat is one, the model will directly return the value of *var*.

var - the controlling parameter value. Input on initializing calls, output thereafter.
 ti - t_i value (0.0). Input.
 tp - t_p value (0.0). Input.
 td - t_d value (1.0). Input.
 k - k value (1.0). Input.
 err - ϵ value. Input.
 derr - derivative of ϵ (0.0). Input.
 cntl0 - set point value of var when ϵ is zero (0.0). Input.
 cntlv - calculated value of var. When stat is equal to one, var is simply set equal to cntlv. When stat is zero, var and cntlv are made equal using a vary-cons loop.
 ub - approximate upper bound on var when used in the vary-cons functions (1.0). Input.
 lb - approximate lower bound on var when used in the vary-cons functions (1.0). Input.

In order to prevent hitting either the upper or lower bounds within the controller vary-cons loop iterations the actual upper and lower bounds used are slightly adjusted from those that are specified by the user. Both bounds are extended by one tenth the distance between the specified bounds and the calculated value of the controlling parameter is truncated at the original user specified bounds.

4.4. Dynamic system tasks

As mentioned within the introduction to the dynamic models, many of the models make use of the task functions for integrating the differential equations of the models and also for representing the mass/momentum exchanges within the system. Each of these will require the user to include within the driver coding a task loop. In the case of the differential equations, the task name used by all of the models is denoted as dyn, thus, the driver code requires a dyn task loop. This also means that the time variable is denoted as dyn.time.

In the case of the mass and momentum exchanges, the models use the task named sta. This task should be set up within the driver coding nested within the dyn task. Since both of these tasks are always required when using the dynamic models, both are defined within the cinit function and thus, do not need to be explicitly allocated using the cnew operator. A generic representation of the dynamic inputs would look something like the following.

```

cinit

"model allocations, parameter specifications, etc. using cnew"

0 tout_increment t_final
  (/dyn.tout exch def
    {dyn.c}
    {{sta.c}
      {
        "model calls, parameter specifications, user define
        vary and cons, etc."
      }
    }
  while
  }
  while
  gass.print mods.print
  "additional output"
  }
  for
  
```

Within the sta task loop the actual number of vary and cons function calls should balance with one variable for each constraint. Thus, the user of the dynamic models must keep track of which models are using a

vary function and which are using a cons function. This is actually not difficult, since each time a new flow path is generated, a vary function is used and each time a flow path is terminated a cons function is used. Thus, vary's are use within the gas and sp functions and cons's are used within the mx and exnz functions. Generally, the parameters that are varied adjust the mass flow rates along the flow path (m in the gas model and sr in the sp model). The constraints are then equations that would define what the mass flow would be (such as choked flow within the exnz model). At times it may be necessary to add within the driver coding additional vary and cons function calls within the sta task loop. This might occur if a model's stat parameter is set to one, implying that the model's vary or cons function is not called. There really is an endless number of possibilities and the user must be clear as to what the problem is that is being set up with the input.

4.5. Dynamic example one

For an example of the use of the dynamic system components we will consider a dynamic analysis of a space nuclear rocket system. The model parameter values for this example are not meant to correspond to any particular system. The example is only for illustrative purposes in showing the type of thinking necessary to use the GPS code for dynamic problems. Figure 2, shows a block diagram for the system. The main thruster nozzle is exnz_1, although part of the flow is diverted to drive the turbo pump, pump_tp, and is exhausted through a second nozzle, exnz_tp. Several valves are provided for system control, a tank shut-off valve, valv_tsov, a pump shut-off valve, valv_psov, a temperature control valve, valv_tcv, and a turbine speed control valve, valv_scv.

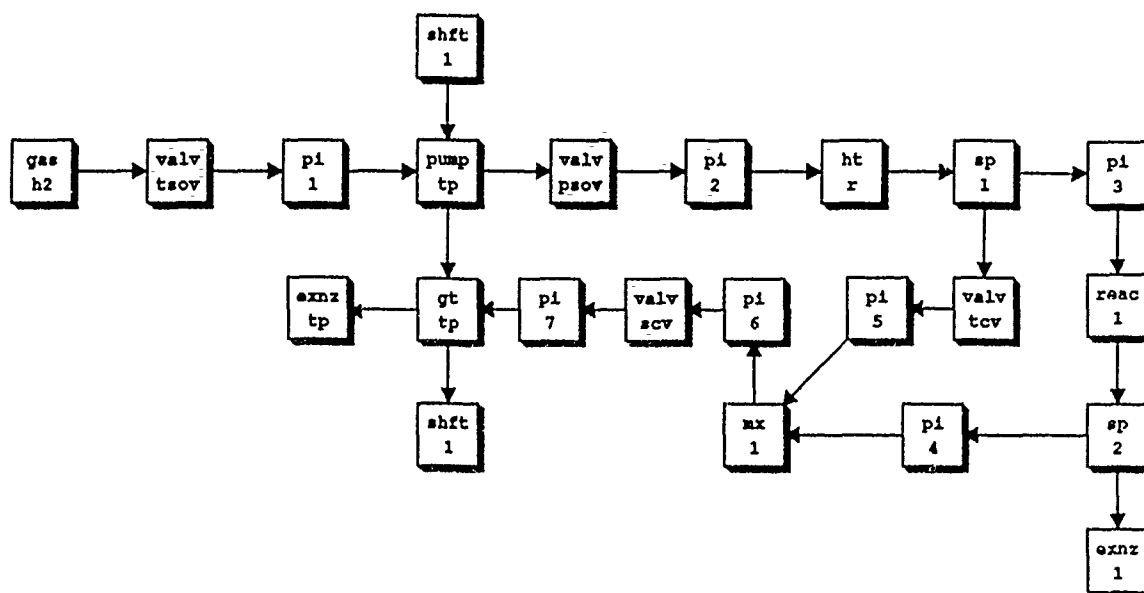


Figure 2.

In analyzing such a system, the first step is to develop a full power design point. This will be used to determine pipe sizings for reasonable flow velocities, exhaust areas for the nozzles, and valve positions. Note, that since this system configuration has one flow initiator and two splitters, there are three automatically imposed parameter variations. Also, since there are two exhaust nozzles and one mixer there are three automatically imposed constraints. As mentioned within the section on the models above, many of the models have a special steady-state option for turning these automatically imposed variations and constraints off. This is done by specifying the model's stat variable to one. With this option specified in the exhaust nozzles, the constraints on mass flow rates (calculated from the flow areas) are not used. Instead, the mass flow rates are treated as the knowns and the flow areas are calculated. This permits one to determine the design flow areas by specifying the design point flow rates. In order to specify this design point flow rate, the stat option will need to be turned on (i.e. set to one) in the gas model representing the hydrogen tank. This will prevent the automatic iterations over the mass flow rate. Note that turning on the stat option within both exhaust nozzles we have turned off two automatically imposed constraints and by turning on the stat option in the tank model we have turned off one automatically imposed parameter variation. Thus, in order to rebalance the number of constraints and parameters being varied, one additional constraint needs to be imposed or another of the parameter variations needs to be removed. Since at the design point there are several other constraints that need to be imposed we will add an additional constraint.

First, at an equilibrium design point, the power required by the pump should be constrained to exactly match the shaft power generated by the gas turbine. This additional constraint will rebalance the number of constraints and the number of parameters being varied. However, as an additional constraint on the system we impose a design value on the gas turbine inlet temperature of 950K. This constraint will be met by varying the temperature control valve pressure drop. By varying this pressure drop, a greater or lesser amount of cold gas flow will be directed to the mixer, thus, affecting the downstream temperature.

The complete input necessary to determine the full power design point is as follows.

```

cinit
[/gas /gas_h2 /id "THR-tH2" /t 20 /p 3.0 /m 1.0 /diam .05 /stat 1 ] cnew
[/valv /valv_tsov /pf 0.02 ] cnew
[/pi /pi_1 /rat_m 1.0 /rat_pf 0.0002 /diam .05 ] cnew
[/pump /pump_tp /rat_rpm 60000. /rat_dp 84.0 /rat_m 1.0 ] cnew
[/valv /valv_psov /pf 0.02 ] cnew
[/pi /pi_2 /rat_m 1.0 /rat_pf 0.0001 /diam .05 ] cnew
[/ht /ht_r /t#1 80. ] cnew
[/sp /sp_1 /sr 0.05 /diam#0 0.05 ] cnew
[/pi /pi_3 /rat_m 1.0 /rat_pf 0.0001 /diam .03 ] cnew
[/reac /reac_1 /tcool 3000. /diam .05 /rat_m 1.0 /rat_pf 0.05 /ucool 1e5 ] cnew
[/sp /sp_2 /sr 0.02 /diam#0 0.05 /diam#1 0.05 ] cnew
[/pi /pi_4 /rat_m .03 /rat_pf 0.0001 /diam .05 ] cnew
[/exnz /exnz_1 /aexp 15e-3 /mach 3.5 /stat 1 ] cnew
[/valv /valv_tcv /pf 0.05276954 ] cnew
[/pi /pi_5 /rat_m .08 /rat_pf 0.0001 /diam .01 ] cnew
[/mx /mx_1 /p 0.92 /diam 0.02 ] cnew
[/pi /pi_6 /rat_m .11 /rat_pf 0.0001 /diam .02 ] cnew
[/valv /valv_scv /pf 0.5 ] cnew
[/pi /pi_7 /rat_m .11 /rat_pf 0.0001 /diam .02 ] cnew
[/gt /gt_tp /rat_pr 1.5 /inertia 0.2 /stat 1 /rat_eff .86282 ] cnew
[/exnz /exnz_tp /diam 1e-2 /stat 1 ] cnew
[/shft /shft_1 /rpm 60000 ] cnew

pump_tp.in gt_tp.in

/sta.prt 2 def /sta.acc 1e-3 def
{sta.c}
{ /valv_tcv.pf valv_tcv.pf 0.02 0.17 vary

```



```

gas_h2.c valv_tsov.c pi_1.c shift_1.c
pump_tp.c valv_psov.c pi_2.c ht_r.c sp_1.c
pi_3.c reac_1.c sp_2.c exnz_1.c
sp_1.s valv_tcv.c pi_5.c mx_1.s sp_2.s pi_4.c mx_1.c pi_6.c
valv_scv.c pi_7.c gt_tp.c exnz_tp.c shift_1.end
/sp_1.sr (pi_7.fl.t-950.) cons
/sp_2.sr (gt_tp.power+pump_tp.power) cons
)
while
gass.print mods.print
/all cdel

```

The model names used in this input correspond to those shown in Figure 2. In order to simplify the specification of the system, many of the model parameters are left unspecified and thus, will assume their default values. After the model parameter specifications, the performance maps for the pump and gas turbine are obtained using the line

```
pump_tp.in gt_tp.in
```

The `sta` task loop is then entered, where the three variables used to control the three constraints are varied. Note, that as discussed above, only the temperature control valve, `valv_tcv`, is explicitly varied. The other variables being varied are defined within the splitters. The system configuration is then coded by calling the models in the appropriate order as defined by Figure 2. The resulting model calls representing the system should be perfectly clear. Finally the constraints are evaluated and the loop terminated. Here only two of the three constraints are explicitly written down. The third one is defined within the mixer (since its `stat` parameter was left as zero). The last line then calls the `gass.print` function and the `mods.print` function to obtain the output.

The resulting output from this code is shown in Appendix D. This Appendix first shows the iterations performed by the equation solver for the `sta` task necessary to establish the three constraints. This output is exactly as with any equation solver task and was explained within the section on the task examples. The rest of the output is obtained from the `gass.print` and `mods.print` functions, and is also the same as with the steady-state examples.

Now consider a start-up run of this same propulsion system. A similar set of inputs as those in design run will be needed but with some minor changes. The final input is as follows.

```

cinit
[/gas /gas_h2 /id "THR-tH2" /t 20 /p 3.0 /m 0.3 /diam .05 ] cnew
[/valv /valv_tsov /pf 0.02 ] cnew
[/pi /pi_1 /rat_m 1.0 /rat_pf 0.0002 /diam .05 ] cnew
[/pump /pump_tp /rat_rpm 60000. /rat_dp 84.0
/rat_m 1.0 /inertia 0.01 ] cnew
[/valv /valv_psov /pf 0.02 ] cnew
[/pi /pi_2 /rat_m 1.0 /rat_pf 0.0001 /diam .05 ] cnew
[/ht /ht_r /t#1 80 ] cnew
[/sp /sp_1 /sr 0.02 /diam#0 0.05 ] cnew
[/pi /pi_3 /rat_m 1.0 /rat_pf 0.0001 /diam .03 ] cnew
[/reac /reac_1 /tcool 100. /tfuel 300.
/diam .05 /rat_m 1.0 /rat_pf 0.05
/ucool 1.0e4 /mfuel 200. ] cnew
[/sp /sp_2 /sr 0.01 /diam#0 0.05 /diam#1 0.05 ] cnew
[/pi /pi_4 /rat_m .03 /rat_pf 0.0001 /diam .05 ] cnew
[/exnz /exnz_1 /aexp 1.5e-02 /mach 3.7 /diam 2.6459e-02 ] cnew
[/valv /valv_tcv /pf 1e-2 ] cnew
[/pi /pi_5 /rat_m .08 /rat_pf 0.0001 /diam .01 ] cnew
[/mx /mx_1 /diam 0.02 ] cnew
[/pi /pi_6 /rat_m .11 /rat_pf 0.0001 /diam .02 ] cnew

```

```

[/valv /valv_scv /pf 0.02 ] cnew
[/pi /pi_7 /rat_m .11 /rat_pf 0.0001 /diam .02 ] cnew
[/gt /gt_tp /rat_pr 1.5 /inertia 0.02 /rat_eff .86282
/rat_cmass 9.8004e-02 /rat_cspped 1.9467e+03 ] cnew
[/exnz /exnz_tp /diam 1.2760e-02 ] cnew
[/shft /shft_1 /rpm 5000 ] cnew
[/cntl /cntl_1 /var 0.02 /cntl0 0.5 /lb 0.02 /ub 0.98 /tp 1.0 ] cnew
/tl 0.0 def

pump_tp.in gt_tp.in

/sta.prt 0 def /sta.acc 1e-3 def /sta.del (-1e-3) def /sta.maxit 10 def
/dyn.prt 2 def

0.0 1.0 30.0
{ /dyn.tout exch def
  (dyn.c)
  {
    (sta.c)
    (dyn.time 15.0 lt
      (/reac_1.power (reac_1.power0+42.442e6*dyn.time/15.0) def)
    if
      /valv_tcv.pf valv_tcv.pf 0.001 0.98 vary
      /valv_scv.pf cntl_1.var def
      gas_h2.c valv_tsov.c pi_1.c shft_1.c
      pump_tp.c valv_psov.c pi_2.c ht_r.c sp_1.c
      pi_3.c reac_1.c sp_2.c exnz_1.c
      sp_1.s valv_tcv.c pi_5.c mx_1.s sp_2.s pi_4.c mx_1.c pi_6.c
      valv_scv.c pi_7.c gt_tp.c exnz_tp.c shft_1.c
      /tl ((0.9*reac_1.fl.t) 950 gt {950.0}) ((0.9*reac_1.fl.t)) -ifelse def
      /sp_2.sr (pi_7.fl.t-tl) cons
      /cntl_1.err ((shft_1.rpm-60000)/20000) def
      cntl_1.c
    }
    while
  }
  while
  gass.print mods.print
}
for
/all cdel

```

First, the stat option for all models should be turned off (or removed from the input). Secondly, the critical areas (or diameters) in the exhaust nozzles as calculated in the design run should be added as inputs. These include exnz1.diam, exnz1.aexp, and exnz_tp.diam. The design point calculated values of the rated corrected mass and corrected speed for the gt_tp model should also be added as inputs. These parameter values basically supply the design point sizing information necessary for those models that are based on modeling that is relative to the design point.

Several other model parameters now need to be changed to reflect initial values for the start-up run. In particular, the reactor gas exit temperature will need to be assigned some nominal starting value, say 100K. Although it is possible to let the code calculate an equilibrium fuel temperature corresponding to this specified gas exit temperature, here an initial 300K fuel temperature will be imposed. An actual system will probably have some pre-flow power level resulting in some temperature somewhat higher than that of the incoming flow. The inlet mass flow rate defined by gas1.m should also be lowered from the design point value. This flow rate will actually be calculated by the code, however, a smaller value, say 0.3, will reflect a more reasonable starting

value for the iterations. The turbo pump shaft speed, defined by `shft1.rpm`, should also be set to some starting value. In the run below, this was set at 5000 rpm. Although, this could be set lower, in general, if the rpm is set to low, insufficient pressure is developed by the pump, and depending on valve positions, temperatures, etc, the flow velocities may become extremely large as the system is initially starting. The flow might actually become momentarily choked at places that the code was not expecting. A lumped component model of a system can only handle choked flow where the model developer intends for it to occur.

Next, the integration loop needs to be added around the `sta` loop. In the run below, the integrations are defined from zero to thirty second, with intermediate output requested at every second.

Finally, the control strategy for the system needs to be considered. This usually refers to the rate of reactor power ramp-up and the rate of opening or closing of valves. For this system, the reactor power ramp-up will be done linearly from some nominal power level, determined by the initial gas temperature rise and defined in the variable `react1.power0`, to the design value of 42.442 MW (plus the initial nominal value) in a time frame of fifteen seconds. The tank shut-off valve and the pump shut-off valve will be instantaneously opened at time zero. The same design point constraint on the inlet gas turbine temperature will be imposed, only now, rather than fixing this temperature at 950K, it will be constrained to be 90% of the reactor outlet temperature up to 950K and constant thereafter. This is done within the inputs by defining the variable `t1` to be this constraining temperature value. Note, that since the temperature out of the mixer is an instantaneous function of its inlet flow conditions, this gas temperature constraint can be meant by using the `vary-cons` operators within the `sta` loop. In a real system, some controller would be needed to actually sense the temperature and gradually adjust the temperature control valve. Finally, to gain some control on the speed control valve, a proportional controller, `cntl1`, is added. Within the `sta` task loop the variable being controlled, `valv_scv.pf`, is assigned the value `cntl1.var`. Upper and lower bounds are also specified for this controlled variable, as well as a set point value of 0.5, corresponding to the design point value of `valv_scv.pf`. The value of the sensor error (i.e. the difference between the variable being sensed and its desired value) is calculated within the `sta` loop by the line

```
/cntl1.err ((shft1.rpm-60000)/20000) def
```

Here, the error will be zero at 60000 rpm, at which point the controller will assign `valv_scv.pf` the value of the set point. The additional dividing factor of 20000 was used instead of adjusting the controller's gain parameter, `k`. Note, this is only a very crude representation of a controller for the speed control valve.

Some of the resulting computer run output is shown in Appendix E. Due to the length of the entire output only the zero, ten, twenty, and thirty second outputs are shown. This output clearly shows that considerably more effort needs to go into the system control strategy even for this very simple example. The reactor fuel temperature using this simple ramp-up has risen to 3343K while the gas temperature has reached only 1118K. Also, the pressure levels are still a long way from the design point.

CHAPTER 5

Thermionic Power System Model Classes

5.1. Introduction

In this chapter we discuss the details of the component models that are used to analysis thermionic power systems. These models were assembled only as a very simple first approximation from existing models furnished by the Air Force. Ultimately, these models will probably be replaced and possibly moved into the steady-state and/or dynamic model class library. At present, these models include the following.

reac -	reactor model
ti -	thermionic converter
rad -	thermal radiator
sp -	power flow splitter
res -	electrical resistor
bc -	boost converter
bus -	electrical bus
mass -	mass calculations

5.2. Thermionic Model Flow Classes

The models used in this collection don't make use of any fluid flow but instead make use of a power-flow. In order to keep the models as simple as possible for this first approximation, only one type of flow class is used and is simply denoted as **flowtype**. **Flowtype** is used to transmit both thermal energy flows and electrical flows. At present, **flowtype** consists of the following variables.

pow -	represents the power being transmitted by the flow in watts.
v -	represents the voltage level in volts relative to ground for electrical flows. Note that for thermal flows v is not used.
i -	represents the current in amps for electrical flows. For thermal flows, i is not used.

At present, the thermionic models are so simple that no special model is used to generate a **flowtype** flow. Instead the **reac** model is used instead. Note that as more modeling details are added to the thermionic model classes the flow structures will also probably need to be changed. The **flowtype** flows are, like all flows, placed on a stack, here denoted as **flows**. **Flows** can be used to print tables of the flows with **flows.print**.

5.3. Thermionic models

In this section we present the details of the thermionic power system models that are presently available. These models are similar to models supplied by the Air Force with some rearranging so that they would fit within the GPS structure. Most of the modeling is composed of simple correlations, some of which need to be replaced with more accurate expressions.

Since the mass of the components is an important consideration in the projected applications of these thermionic systems, a **masstype** class was also included with the models. This **masstype** class is used to store the various component masses and, in some cases, sub-component mass. The **masstype** structures are stored on the stack **masss**, which is then accessed by the mass model to calculate the total system mass. **Masstype**, at present, consists of only the single variable,

mass -	representing the mass of the corresponding object in kg.
--------	--

5.3.1. Reactor (reac) model

The reactor model is used to initiate two flows. The first is the flow utilized by the thermionic converter and the second is the waste heat flow. The split of the total reactor generated power between these two flows is given by an input efficiency. Technically, this efficiency represents the thermionic conversion efficiency, but is included within this model rather than the thermionic model so that upstream model references are avoided.

The main calculations are performed within the model's c function and consist mainly of mass and sizing calculations. The following calculations are performed.

$$Pow_1 = \eta Pow$$

$$Pow_2 = (1 - \eta) Pow$$

$$r_{rc} = 0.13 \quad 10^4 \leq Pow \leq 2 \times 10^4$$

$$= 0.20 \quad 2 \times 10^4 < Pow < 4 \times 10^4$$

$$h_{core} = 2.81 Pow_1 + 0.21$$

$$m_{core} = 24.2 Pow_1 + 23.5$$

$$m_{ss} = 100$$

$$m_{rs} = 15.8 Pow_1 + 149 \quad d_{sep} \geq 15.0$$

$$= 21.4 Pow_1 + 262 \quad 15 > d_{sep} \geq 10.0$$

$$= 26.9 Pow_1 + 374 \quad 10 > d_{sep}$$

$$r_{rs} = 1.56 Pow_1 + 0.91$$

$$V_{rs} = 1/3 \pi h_{rs} (r_{rs}^2 + r_{rc}^2 + r_{rs} r_{rc})$$

$$l_{boom} = d_{sep} - 1$$

$$m_{boom} = \rho_{boom} l_{boom}$$

where, Pow is the input reactor power level, Pow_1 is the power used in the thermionic component, Pow_2 is the rejected waste heat, r_{rc} is the reactor core radius, r_{rs} is the radiation shield radius, h_{core} is the reactor core height, m_{core} is the mass of the core, m_{ss} is the mass of the safety systems, m_{rs} is the mass of the radiation shield, m_{boom} is the mass of the boom, V_{rs} is the volume of the radiation shield, and l_{boom} is the length of the boom, and d_{sep} is the separation distance between the shield and core. The c function also outputs to the flow stack the flow that is to be sent to the thermionic converter. The model's s function can be called to obtain the waste heat flow.

The model has the following variables.

pow -	reactor power level (1e6 watts). Input.
eff -	thermionic conversion efficiency (0.13). Input.
sep -	separation distance d_{sep} (10 m). Input.
radius -	core radius r_{rc} (m). Output.
height -	core height h_{rc} (m). Output.
rhboom -	boom density ρ_{boom} (10.0 kg/m). Input.
lboom -	length of the boom l_{boom} (m). Output.

radiusrs -	radius of the shield r_{rs} (m). Output.
heightrs -	height of the shield h_{rs} (0.37 m). Input.
volrs -	volume of the shield V_{rs} (m^3). Output.
mcors -	mass of the core m_{core} (kg). Output.
mss -	mass of the safety systems m_{ss} (kg). Output.
mrs -	mass of the radiation shield m_{rs} (kg). Output.
mboom -	mass of the boom m_{boom} (kg). Output.
fl -	primary flow to the converter. Output.
fls -	secondary or waste heat flow. Output.

Note that each of the mass variables is really one of **masstype** and thus, for example, mcors would be referred to as mcors.mass within the GPS inputs. Similarly, both of the output flows are of **flowtype**. Thus, the output power to the converter would be referenced as fl.pow.

5.3.2. Thermionic Converter (ti) model

The thermionic converter model takes a power flow and partitions it into an I-V character. The model's calculational function **c** requires one flow on the flows stack and puts one flow back onto that stack. The following calculations are performed.

$$n_{cs} = v / v_{conv}$$

$$i = Pow / v$$

$$n_{cp} = i / i_{conv}$$

$$m = 13.15Pow - 3.5$$

where v is the specified output voltage from all converters, v_{conv} is the individual converter output voltage, Pow is the input power, i is the output current, i_{conv} is the individual converter output current, n_{cs} and n_{cp} are the number of converters in series and parallel, respectively, and m is the total converter mass.

The model has the following variables.

v -	total output converter voltage (250 V). Input.
vconv -	individual converter output voltage v_{conv} (0.67 V). Input.
iconv -	individual converter output current i_{conv} (62.0 amps). Input.
ncs -	number of converters in series n_{cs} . Output.
ncp -	number of converters in parallel n_{cp} . Output.
m -	total mass of the converter (kg). Output.
fl -	output electrical flow from the converter. Output.

5.3.3. Radiator (rad) model

Rad models a thermal radiator. The model requires one flow from the flow stack and puts one flow back onto the stack. The calculations are as follows.

$$A = \frac{Pow}{\epsilon \sigma (T^4 - T_{space}^4)}$$

$$m = \rho A$$

where Pow is the input power (heat), ϵ is the surface emissivity, σ is the Stefan-Boltzmann constant, A is the radiator surface area, T is the surface temperature, T_{space} is the effective space temperature, ρ is the radiator mass per unit surface area, and m is the total radiator mass.

The model has the following variables.

t -	radiator surface temperature (K). Input.
t _{space} -	effective space temperature (255K). Input.
rho -	radiator density per unit area (44.0 kg/m ³). Input.
e -	surface emissivity ϵ (0.85). Input.
area -	surface area (m ²). Output.
m -	radiator mass (kg). Output.
fl -	output flow.

5.3.4. Boost Converter (bc) model

The boost converter models a repartition of an input power flow into new I-V characteristics at a specified input efficiency. The model's calculational function *c* requires one input flow from the stack and puts one flow back onto the stack. The calculations are as follows.

$$Pow_e = \eta Pow$$

$$Pow_w = (1 - \eta) Pow$$

$$i = Pow_e / v$$

where *Pow*, *Pow_e* and *Pow_w* are the input, electrical output, and waste heat output power flows, η is the converter efficiency, *v* is the specified converter output voltage and *i* is the converter output current. The waste power flow is obtained by calling the secondary *s* function of the model.

The model has the following parameters.

v -	output voltage (250 V). Input.
eff -	converter efficiency (0.95). Input.
fl -	output electrical flow.
fls -	output waste heat flow.

5.3.5. Flow Splitter (sp) model

The flow splitter model is used to split a single power flow into two flows based on an input split ratio. For electrical flows this split ratio can be thought of as a current split ratio and for thermal flows as a power split ratio. The model's calculational function *c* requires one input flow and generates one output flow. The second flow is then obtained from the model's secondary function *s*. The calculations are as follows.

$$i_2 = sr \ i_{in}$$

$$v_2 = v_{in}$$

$$i_1 = (1 - sr) i_{in}$$

$$v_1 = v_{in}$$

$$Pow_2 = sr \ Pow_{in}$$

$$Pow_1 = (1 - sr) Pow_{in}$$

where *sr* is the split ratio, *i*, *v*, *Pow* represent the current, voltage, and power, respectively, and the subscripts, *in*, 1, and 2 correspond to the input, and two output flows.

The model has the following variables.

- sp - split ratio (0.1). Input.
 fls - second or split off flow. Output.
 fl - remaining flow. Output.

5.3.6. Resistor (res) model

The resistor models an electrical resistance and hence, an electrical voltage drop. The model's calculational function requires one flow from the stack and outputs one flow back to the stack. The calculations are as follows.

$$v = v_{in} - i_{in} r$$

$$i = i_{in}$$

$$Pow = vi$$

where v_{in} is the input voltage, v is the output voltage, i_{in} is the input current, i is the output current, r is the resistance, and Pow is the output power.

The model has the following variables.

- r - electrical resistance (0.0 Ohms). Input.
 fl - output flow.

5.3.7. Bus (bus) model

The bus model, at present, does nothing, i.e. its only output flow is exactly as the input flow.

5.3.8. Mass (mass) model

The mass model is used to perform a global sum over all the stored **masstype** variables in all the models. In addition, it is used to furnish an electrical distribution system mass and an instrument and control system mass. Both of these, at present, are simply taken as fixed numbers.

$$m_{dist} = 220$$

$$m_{ic} = 222$$

Note that since this model requires information that is calculated in all the other models, it should only be called after all the other models have been called. The print function for the mass model will generate a table of the masses of all the components.

The model has the following variables.

- mic - mass of the instrument and control subsystem (kg). Output.
 mdist - mass of the electrical distribution subsystem (kg). Output.
 mtot - total system mass (kg). Output.

5.4. Thermionic System Example

In this section we present a very simple thermionic power system example using GPS. The system diagram for the example is shown in Figure 3. The example consists of a reactor (react_1) driving a thermionic converter (ti_1). The power flow from the converter is then partially split off (sp_shunt) into a shunt radiator (rad_shunt) with the rest of the power flow going through a resistance (res_ti), a boost converter (bc_1), another resistance (res_bc) and finally into the power bus (bus_1). The reactor's waste heat is then feed into the primary radiator (rad_prim), the split off flow from sp_shunt is then feed into the shunt radiator (rad_shunt) and finally, the boost converter's waste heat is feed into another radiator (rad_bc).

One system constraint will also be imposed and that is to produce exactly 40 kw at the bus. The total input necessary to run the problem is as follows.

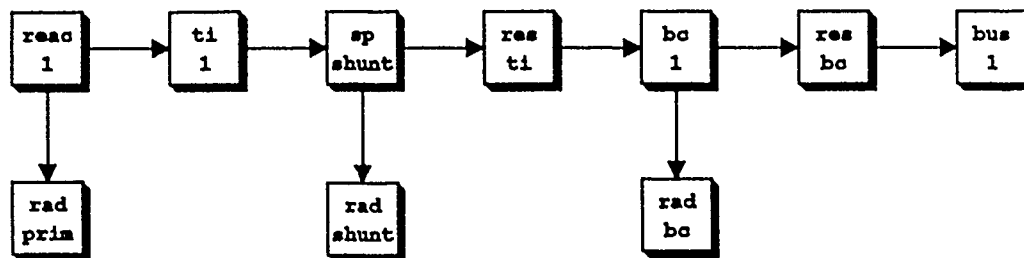


Figure 3.

```

cinit
[/reac /reac_1 /pow 100e4] cnew
[/ti /ti_1 /v 6] cnew
[/sp /sp_shunt /sr 0.01] cnew
[/bc /bc_1 /v 100 /eff 0.85] cnew
[/res /res_ti /r 1e-4] cnew
[/res /res_bc /r 1e-4] cnew
[/rad /rad_prim /t 1000] cnew
[/rad /rad_shunt /t 400] cnew
[/rad /rad_bc /t 400] cnew
[/bus /bus_1] cnew
[/mass /mass_sys] cnew
[/task /a] cnew

{a.c}
  (/reac_1.pow 80e4 40e4 120e4 vary
   reac_1.c ti_1.c sp_shunt.c res_ti.c bc_1.c res_bc.c bus_1.c
   reac_1.s rad_prim.c
   sp_shunt.s rad_shunt.c
   bc_1.s rad_bc.c
   /reac_1.pow (bus_1.fl.pow-40e3) cons
  )
while
mass_sys.c flows.print mods.print
/all cdel

```

The output for this GPS input is shown in Appendix F. Note, that this output should not be considered a typical thermionic system as not all of the parameters had reasonable values assigned to them. This example is only meant to show what a typical input and its resulting output would look like.

CHAPTER 6

Graphics

6.1. Introduction

The graphics currently available within GPS are only preliminary but are sufficient to generate simple two-dimensional plots and system diagrams of component layouts. The two-dimensional plots are implemented using a model class, a networking communications package, and the NeWS toolkit wire server. The system diagrams also make use of the wire server and thus, the graphics only are available when this wire server is available.

6.2. Two-dimensional Plots

Two-dimensional plots of arbitrary user selected independent (x values) and dependent (y values) variables are generated by using a model class denoted as **plot**. For each plot desired an instance of this plot class should be generated using **cnew**. When the plot class instance is generated a new window will pop open on the screen. Initially, the window will be blank with only the label specified. The label will be the same as the plot class instance name.

The variables that can be defined for the plot model are as follows.

- xl - character string representing the x-axis label ("x-axis"). Input.
- xb - lower bound of the independent variables (0.0). Input.
- xub - upper bound of the independent variables (1.0). Input.
- yl - character string representing the y-axis label ("y-axis"). Input.
- yb - lower bound of the dependent variables (0.0). Input.
- yub - upper bound of the dependent variables (1.0). Input.

At present, the increment used along each axis is one fifth of the total axis length.

The data for each plot is obtained by using the **c** function for the class. This function requires arguments and, as such, will need to be called using the call operator. The arguments are nothing more than the x,y pairs of data to be plotted. Thus, one would write

[x y] /plot1.c call

to plot the x,y pair in plot1. The plots generated use straight line segments between the plotted points.

At present there is no delay between popping open a window and continuing the execution of the GPS input. Thus, since the act of popping open a window may take some time, it is possible for very simple problems that the entire GPS input may have been executed before the plot window has been opened. No data is lost in this case, as the data going to the plot window is stored and simply plotted when the window becomes opened. At present, only 400 x,y pairs are stored per window. A check is made when doing the plotting that the new x,y pair is at least one pixel different than the previous x,y pair. Thus, 400 values are usually sufficient for most plots. Data is also properly stored if a plot window is closed.

As the plot windows are based on the OpenLook windows, these plot windows can be moved, resized, closed, and open. The resizing, however, does not resize the plot itself. Any damage to the window is automatically repaired from the data stored for the plot. As the window is receiving information from both the GPS code via the communications package and through the mouse interactions, the plot windows are implemented as separate processes. Thus, one can quit a GPS session and any plot windows generated will remain. These windows can be terminated by using the quit item of the windows frame menu.

The following GPS input is an example of the use of the plot class .

```
cinit
[/plot /a /x1 "x label" /xub 2.0] cnew
[/plot /b /x1 "x stuff" /xub 2.0 /y1 "z stuff"] cnew
[/plot /c /xub 2.0 /y1 "uavg"] cnew
0 0.1 2.0
  {/x exch def /y (0.25*x*x) def /z (exp(-x)) def
  [x y] /a.c call
  [x z] /b.c call
  [x (x*exp(-x))] /c.c call
  }
for
/all cdel
```

Here three plot windows will be popped open showing in window "a" a plot of $x^2/4$ versus x from 0 to 2.0, in window "b" a plot of e^{-x} versus x , and in window "c" a plot of xe^{-x} versus x .

6.3. System Diagrams

As mentioned previously, GPS can be used to create system diagrams, such as those used in Figures 1, 2 and 3. This is done by using the `mods.config` function. For each collection of components there is a C function denoted as `modsconfig`, which will read a GPS input file, parse it into the components that are being used, and then pop open a window in which a very simple linear representation of the system appears. This simple system diagram can then be edited into a reasonable representation of the system under consideration.

The next section discusses the GPS inputs necessary to pop open this configurations window followed by a section that discusses the editing of the diagram using the mouse. This feature of generating the system diagrams requires the use of the SUN NeWS window environment.

6.3.1. Configuration Windows

The `mods` stack class generally has a `config` function which when called will generate a configuration file and then pop open a NeWS window in which the system configuration diagram can be edited. This function requires one character string argument representing the name of the GPS input file. Thus, in order to generate the configuration window, one must first initialize the `mods` stack class using a call to `cinit` and then use the call operator to call the `mods.config` function with the file argument. Thus, the GPS input necessary would look as follows.

```
cinit ["file.dat"] /mods.config call
```

Here "file.dat" is the name of the GPS input file representing the system for which a diagram is required. Note that this input file is noting special, just the typical input that was specified in the examples previously given. System constraints, parameter sweeps, optimizations, etc, are all ignored by the `mods.config` function as they are not pertinent to the generation of the system diagram.

Once `mods.config` has been called the mouse cursor will change to a '+' indicating to the user to size a window (using the middle mouse button). Initially, the window will display the system diagram as a series of linear flow paths, some of which may be longer than that of the window. The editing of the flow paths into a reasonable looking diagram is done entirely with the mouse and the left and middle mouse buttons. Once the user is satisfied with the diagram the right mouse button can be used to pop open the main menu.

Only four menu items exist. The first is to save the diagram. If selected the diagram is saved into a file denoted as "file.conf", where "file" is the same name as used originally in the GPS `mods.config` call. The second item is for repainting the diagram in case of damage that has not automatically been repaired. The third item is to print the diagram on the printer. This item also generates the file named "gsalt.prt". This file is the PostScript code that was used to generate the diagram on the printer and can be resaved and edited manually if desired. Note that "gsalt.prt" is overridden each time a new diagram is printed. Finally, the last menu item is for quitting the diagram window.

The call to `mods.config` returns almost immediately, since it really generates a new UNIX's process for dealing with the system diagram editing. Thus, it is possible to actually place the call to `mods.config` in the

same file that is being called, although, it is probably more useful to manually type the `mods.config` call and generate a diagram before actually running the system problem using GPS. This is because the diagram clearly shows, even though it is not well laid out, the model connectivity that the GPS inputs have defined. Thus, the diagram might show an error that has been made in defining the configuration with the model calls.

Once a diagram has been generated and saved for a given input file, this diagram is used the next time `mods.config` is called. This is true even if the new GPS file has been edited to a slightly different configuration possibly with a greater or lesser number of models. The `mods.config` will attempt to merge the new configuration with the old so that less diagram editing will be required to generate a reasonable looking diagram the next time. It should be noted, however, that the diagram requires layout information that is not provided when using GPS to analyze a system and thus, when new models are added to a configuration some strange looking diagrams may initially appear. However, during the diagram editing no models can be added or removed and the connectivity of the flows between the models can not be altered in any way, only the layout can be altered. In other words, the diagram can only represent what was set up within the GPS inputs.

6.3.2. Diagram Editing

As mentioned previously, all the editing of a diagram is done using the mouse. Basically, there are only two main ideas in editing a diagram. The first is moving the component models around and the second is anchoring one or more of the models so that they will not be moved as others are. The moving of components is accomplished using the middle mouse button and the anchoring is done using the left mouse button. In each case, the models of a flow path are affected. By "flow path", we mean a collection of models that are passed through by a single flow. The first model in the path will generate the flow and the last will generally, terminate the flow. Thus, for example, in the dynamic models, the `gas.c`, `sp.s`, and `shft.c` model calls generate flows and the `mx.s`, `exnz.c`, and `shft.end` model calls will terminate flows.

Assuming first, that no anchored models have been specified, by placing the mouse cursor on a model, pressing and holding the middle mouse button, and then dragging the mouse to a new point, the pointed to model is translated rectilinearly to the new location. All other models within the flow path that this model was in are also translated by exactly the same vector translation. No rotations or distortions occur.

By placing the mouse cursor on a model and clicking (press and release) the left mouse button, the pointed to model is anchored. This is indicated by a small '+' sign appearing within the model's box. The anchor can be changed by repeating the procedure on another model. If the left mouse button is clicked on the background no model will be anchored. This anchor will be called the primary anchor. At any one time only one model can have a primary anchor. Once a model is anchored in a flow path, the translation that is done using the middle mouse button is altered in the following way. Those models that appear between the primary anchored model and the translated model undergo a rotation, those models that are after the translated model undergo the usual rectilinear translation. By "after" we mean those models that are nearest the translated model but not between the translated and primary anchored models. Thus, "after" could also mean "before" the translated model in the sense of the passage of the flow through the models.

At times it is necessary to anchor two models within a flow path, this is accomplished by pointing to the first model, pressing the left mouse button, dragging the mouse to the second model, and then releasing the left mouse button. Both the models should then have the anchor signs. The model pointed to first, as before, will be the primary anchor and the second one will be called the secondary anchor. In this case, the translations are affected as follows. Those models between the primary anchor and the translated model undergo a rotation, while those between the translated model and the secondary anchor undergo a translation. If the translated model is not between the anchored models, the models between the translated model and the closest anchor (as measured along the flow path) undergo the rotation and the models "after" the translated model undergo a translation.

When two models are placed on top of each other they will fuse into a single box, and a small "o" will appear within the box indicating that there are overlaid models at this point. Overlaid models also act as if they were anchored models. Small movements of such overlaid models can only be done by moving one model completely away from the other, making the small adjustment, and then moving the overlaid model back onto the adjusted model. Note that even after a model has had other overlaid models moved away from it, and thus effectively removing its overlaid status, the overlay status is not changed until that model itself is moved.

The primary anchor is also used to clean up vertical and horizontal flow path lines. This is done by anchoring a model and then clicking the middle mouse button on some other model approximately vertical or horizontal to the anchored model. When this is done, the adjusted model will jump to exactly vertical or horizontal position relative to the anchored model. In this case the adjusted model does not have to be within the same flow path. Thus, models in different flow paths can be aligned relative to each other.

Initially, the flow paths between the models are straight line paths. This at times, is not sufficient, since flow paths may need to make turns, etc. A turn or kink in the flow path, is generated by simply pointing to a model, pressing the left mouse button and dragging the mouse to the very next model in the flow path. When this is done, the flow path will show a small circle midway between the two models. This small circle can be anchored and moved just like any other model in the system. For ease of use, however, the movement of the circle only affects that circle. That is, movement of the circle is as if both models on either side of the circle were anchored. Additional kinks can be generated by pointing to the model or circle with the left mouse button and again dragging the mouse to the next model or circle in the flow path and releasing the button. The kinks can be removed by pointing to the kink with the left mouse button and dragging the mouse to the previous flow model in the path (which may, of course, be another kink) and then releasing the mouse button.

CHAPTER 7

GPS Model Interfacing

7.1. Introduction

In general GPS was designed to be able to reference both model class functions and model class data structure elements by name. In addition, GPS was designed to be as generic as possible and not refer internally to any specific class type. In this way GPS could simply be linked with any suitable class library without the need to make any changes to GPS itself.

In order to accomplish these goals, each model class instance needs a way of returning the location of a class member given a literal reference for that member. To do this, each class has a substructure denoted as *refee* which contains three pointers, one to the class instance data structure itself, one to the name of the instance, and one to a reference function for the class. This *refee* substructure is then placed on a stack, denoted as *cstack*. GPS then references this *cstack* to locate any particular model instance, which in turn, given the location of the reference function for the class. This reference function can then be called to either locate a particular model instance's variable or call a particular model function. More details on the model reference function will be presented below.

New models are easily added to the any of the component libraries since each model is essentially self contained. The only interfacing with the GPS coding is through the new model class instance allocator and the model's *ref* function. However, in order to retrieve flows and make use of the property codes the model will need to make appropriate calls to the flow and property class member functions. Once a new model is developed, it is simply compiled and added to the appropriate component library.

In general, a model may contain most any C language coding that is necessary to describe the model's phenomena. However, because models are called with some of their input parameters perturbed slightly for evaluation of derivatives used in the mathematical utilities, the models must represent true functions of the inputs. Thus, the same outputs should be obtained from the model for each call using the exact same input flows and parameter values. Note that this precludes using some modeling parameter that uses a value from a previous call to the model, unless that parameter is simply used as an initial guess to some iteration. In that case these internal model iterations should converge to a tolerance that still permits evaluating model parameter derivatives by finite differencing. Internal iteration convergence should be kept fairly tight. It does not help to speed up the code by loosening the convergence criteria, since this will often result in more iterations being used by the driver coding in solving system constraints.

7.2. Interfacing example

In order to describe the details of adding a new model, we will go through the steps of adding a fictitious model to the dynamic model library. Let us suppose this model is called *xmod* and requires one *gastype* flow and one *shfttype* flow as inputs, both of which are also output flows. Further, let us suppose the model has two parameters, *parm1* and *parm2*, both double precision variables and that the *parm2* parameter is governed by the equation

$$\frac{dparm2}{dt} = f(parm1, parm2, \dots)$$

where the function *f* will be left unspecified, but, of course, would be known for some actual model. The steps in developing this model would amount to defining a model class and its member functions. The model class for this new model would be as follows.

```
struct xmod
{char name[16];
 struct refee *z;
```

```

double parm1, parm2;
struct gastype fl;
struct shftype shftf;
);
struct refec *xmodnew();
void *xmodref()

```

The first variable, name should always be included. It is used to store the name of the model for use in printouts. Next, the structure refec is used to locate this particular model's variables and member functions by the GPS code. Its use and structure will be described later. Following the refec structure the two double parameters are declared followed by the declarations of the gastype and shftype flows, where we have used for their names fl and shftf, respectively. These names are, of course, anything the modeler wishes to use.

The model's class member functions can be anything the model developer requires. However, two functions must be provided. The first is an allocator function, recognized by having the same name as the model class with the suffix new, which takes as an argument a character string representing the model's name. This function must return a pointer to the structure refec which is defined in the header file util.h.

The second is the ref function which will be used by the GPS code to reference the models variables and functions.

Once this xmod class declaration is defined, one needs to code up the member functions.

The new function is called whenever an instance of the class is required and is also the place where default values for the model parameters can be defined. For our xmod model we would have the following.

```

xmodnew(char *s)
{struct xmod *z;
 z=(struct xmod*)calloc(1,sizeof(struct xmod));
 strcpy(z->name,s); z->z.spt=(void*)z; z->z.namp=z->name; z->z.ref=newref;
 parm1=10.; parm2=20;
 stackput(mods,(void*)&z->z,(void*)z->name);
 return(&z->z);
}

```

Here an instance of the xmod structure is allocated and the elements of the refec substructure and the variable name are assigned values. The refec structure contains the three variables listed, spt, which points to the newly allocated xmod structure, namp which points to the model's name and finally, ref, which is a pointer to the model's ref function. The next line gives the defaults to the two model parameters. The coding here would be different for each model and can be most any type of initializations the modeler needs to make. Finally, the model's allocated structure is placed on the mods stack by placing the address of the substructure refec on the stack using the stackput function. The function terminates by returning the address of this refec substructure. Note that this coding with the exception of different ref functions and model parameters would be the same for every new model.

The calculational function for this xmod class is as follows.

```

void xmodc(z)
struct xmod *z;
{double f;
 z->fl=*gasget(); z->fl.namp=z->name;
 z->shftf=*shftget(); z->shftf.namp=z->name;
 if (dyn->state==0)
 {
 /* do any initialization calculations */
 }
 /* evaluate function f */
 diff( &z->parm2, f, &dyn );
 /* evaluate exit flows */
 z->fl = ...
 z->shftf= ...
}

```

```

    stackput(gass,(void*)&z->fl,0);
    stackput(shifts,(void*)&z->shftf,0);
}

```

Here we declare a double precision variable *f* to store the value of the time derivative of *parm2*, then we obtain the *gastype* and *shfttype* flows. *Gasget* simply retrieves a *gastype* flow from the *gass* and *shftget* retrieves a *shfttype* flow *shifts*. After obtaining these flows their *namp* variable is assigned the model's name. In this way the model name is associated with the flow for printout purposes.

If calculations are required before the integrations over time begins they can be placed within a conditional block testing the *dyn->state* variable for zero. This variable is only zero before the integrations start.

The modeling calculations are then coded. At some place within this modeling a call to *diff* will be required to represent the *xmod*'s differential equation and the exit values for both the *fl* and *shftf* flows will need to be calculated. These are then put back onto their respective flow stacks using the *stackput* function.

The model probably should have a *print* function, which could be coded as follows,

```

void xmodprint(z)
struct xmod *z;
{printf("\n%-12s parm1=%e parm2=%e",z->name,z->parm1,z->parm2);
}

```

The model's reference function is denoted by the suffix *ref*. The *ref* function takes three arguments. The first is a pointer to the model class's structure, the second is a character-string argument representing a member variable name, and the third is a char argument representing a variable type. For variables that are double precision, integer, or character strings, type is returned as either a 'd', 'i', or 's'. For member functions type is returned as 'f'. These are the values displayed in Table 3. In the each of these cases, *ref* also returns a pointer typed cast to (void*) representing the address of the variable or, in the case of a function returning no values, a null pointer. Additionally for functions, the *ref* function will also call the function if type is specified on input to be 'z'. If the function has a returned value, type is then reassigned as either a 'd', 'i', or 's'. For this *xmod* model the *ref* function would look like the following.

```

void *xmodref(z, st, type)
struct xmod *z; char *st, *type;
{if (strcmp(st,"c")==0)
    {if (*type=='z') xmodc(z); *type='f'; return 0;}
  else if (strcmp(st,"print")==0)
    {if (*type=='z') xmodprint(z); *type='f'; return 0;}
  *type='d';
  if (strcmp(st,"parm1")==0) return (void*)&z->parm1;
  else if (strcmp(st,"parm2")==0) return (void*)&z->parm2;
  else
    {printf("\n%s.%s unknown",z->name,st); exit(-1);}
}

```

Here *xmodref* will either call the calculational function or print function or return the locations of *parm1* or *parm2*.

7.3. Other requirements

Besides these requirements on the individual C model classes, the GPS code expects that two functions will be provided as externals. These two functions are called *cnew* and *cinit*. Function *cnew* takes two arguments. The first is a character string pointer specifying the name of a C model class type and the second is a character string pointer specifying the instance name of that class type. The function *cnew* should call the new function of the appropriate class type returning the same pointer returned by that model's new function. It should also put that new model's *refee* substructure on the *cstack*. Note that *cnew* is the C function that is used by the GPS *cnew* operator to allocate C model classes. Thus, any class that the GPS is to communicate with must be recognized by the *cnew* function.

The `cinit` function takes no arguments and returns a void. This function is called by the GPS `cinit` operator and should allocate, by calls to `cnew`, any C model classes that must exist for the current collection of classes that are being linked to GPS. An example of such a class is the `mods-stack` class instance for storing the model instances. `Cinit` may also perform any other initialization tasks needed by this collection of classes.

References

1. H.Geyer and G Berry, "The Systems Analysis Language Translator (SALT): User's Guide," ANL/FE-85-3, ANL, 1985.
2. H.Geyer and G.Berry, "The Systems Analysis Language Translator (SALT): Programmer's Guide," ANL/FE-85-4, ANL, 1985.
3. PostScript Language Reference Manual, Addison-Wesley Publishing Co. 1987.
4. Powell, M.J.D., "A Hybrid Method for Nonlinear Equations," in Numerical Methods for Nonlinear Algebraic Equations, Gordon and Breach Science Publishers, New York, 1970.
5. Powell, M.J.D., "A Fast Algorithm for Nonlinearly Constrained Calculations," presented at the 1977 Dundee Conf. on Numerical Analysis, Dundee, U.K., 1977.
6. Gear, C.W. "Numerical Initial Value Problems in Ordinary Differential Equations," Prentice Hall, Englewood Cliffs, N.J., 1971.

APPENDIX A

Task Class examples

This appendix shows each of the example problems described in section four and their corresponding output. Lines in *italics* are not part of the inputs or outputs but were added to label and explain things.

EXAMPLE ONE

```
cinit
[/task /a] cnew
{a.c)
  {/x 1.0 0.0 2.0 vary
   /x (x*x-exp(-x)) cons
  }
while
/a cdel

task: a  n=0  f=6.321206e-01
      x= 1.000000e+00
      c= 6.321206e-01
      h= 7.1266e-02 hs= 7.1266e-02 mu=0.00e+00 n=7.13e-02 s=7.13e-02 a=1.00e+00

task: a  n=1  f=5.690847e-02
      x= 7.330436e-01
      c= 5.690847e-02
      h= 5.7761e-04 hs= 5.7761e-04 mu=0.00e+00 n=6.98e-04 s=6.98e-04 a=1.00e+00

task: a  n=2  f=6.026605e-03
      x= 7.066324e-01
      c= 6.026605e-03
      h= 6.4778e-06 hs= 6.4778e-06 mu=0.00e+00 n=9.79e-06 s=9.79e-06 a=1.00e+00

task: a  n=3  f=6.978994e-05
      x= 7.035041e-01
      c= 6.978994e-05
```

EXAMPLE TWO

```

cinit
[/task /a] cnew
{a.c}
  (/x 2 (-20) 20 vary
   /y 2 (-20) 20 vary
   /z 2 (-20) 20 vary
   /x (pow(x-1,2)-y) cons
   /y (y-2*log(exp(x)+1)) cons
   /z (z*z-x) cons
  )
while
"\nx=%0.2f y=%0.2f z=%0.2f" [x y z] printf
/all cdel

task: a n=0 f=2.241267e+00
x= 2.000000e+00 2.000000e+00 2.000000e+00
c=-1.000000e+00 1.525738e-01 2.000000e+00
h= 5.2328e-01 hs= 5.2328e-01 mu=4.55e-01 n=7.17e-01 s=1.68e-01 a=6.30e-01

task: a n=1 f=4.511650e-01
x= 2.603536e+00 2.215696e+00 1.690980e+00
c= 3.556303e-01 -1.077238e-01 2.558760e-01
h= 4.7315e-02 hs= 4.7315e-02 mu=0.00e+00 n=3.66e-02 s=1.37e-02 a=7.23e-01

task: a n=2 f=1.463562e-01
x= 2.442031e+00 2.197887e+00 1.589939e+00
c=-1.184341e-01 4.319375e-03 8.587682e-02
h= 3.9862e-03 hs= 3.9862e-03 mu=0.00e+00 n=3.81e-03 s=8.62e-04 a=6.27e-01

task: a n=3 f=1.840560e-02
x= 2.491854e+00 2.233754e+00 1.583782e+00
c=-8.126091e-03 2.959046e-04 1.651197e-02
h= 3.3636e-05 hs= 3.3636e-05 mu=0.00e+00 n=3.47e-05 s=7.34e-06 a=6.93e-01

task: a n=4 f=1.664113e-03
x= 2.495687e+00 2.236516e+00 1.580270e+00
c= 5.629013e-04 -1.760654e-05 1.565920e-03
h= 2.3278e-07 hs= 2.3278e-07 mu=0.00e+00 n=3.21e-07 s=6.46e-08 a=8.22e-01

task: a n=5 f=2.549055e-04
x= 2.495456e+00 2.236347e+00 1.579781e+00
c= 4.208626e-05 -1.330936e-06 2.514037e-04

x=2.50 y=2.24 z=1.58

```

EXAMPLE THREE

```

cinit
[/task /a ] cnew
[/task /b ] cnew
/z 2.0 def
{a.c)
  {/x 2.0 (-20) 20.0 vary
   /y 2.0 (-20) 20.0 vary
   {b.c)
     {/z z (-20) 20 vary
      /z (z*z-x) cons
     }
   while
     /x (pow(x-1,2)-y) cons
     /y (y-2*log(exp(x)+1)) cons
   }
while
"\nx=%f y=%f z=%f" [x y z] printf
/all cdel

task: b n=0 f=2.000000e+00
x= 2.000000e+00
c= 2.000000e+00
h= 2.5000e-01 hs= 2.5000e-01 mu=0.00e+00 n=2.50e-01 s=2.50e-01 a=1.00e+00

task: b n=1 f=2.500001e-01
x= 1.500000e+00
c= 2.500001e-01
h= 3.9063e-03 hs= 3.9063e-03 mu=0.00e+00 n=5.10e-03 s=5.10e-03 a=1.00e+00

task: b n=2 f=4.081634e-02
x= 1.428571e+00
c= 4.081634e-02
h= 1.0412e-04 hs= 1.0412e-04 mu=0.00e+00 n=1.94e-04 s=1.94e-04 a=1.00e+00

task: b n=3 f=1.189769e-03
x= 1.414634e+00
c= 1.189769e-03
h= 8.8472e-08 hs= 8.8472e-08 mu=0.00e+00 n=1.75e-07 s=1.75e-07 a=1.00e+00

task: b n=4 f=6.007310e-06
x= 1.414216e+00
c= 6.007310e-06

task: a n=0 f=1.011572e+00
x= 2.000000e+00 2.000000e+00
c=-1.000000e+00 1.525738e-01

task: b n=0 f=5.807310e-06
x= 1.414216e+00
c= 5.807310e-06

task: b n=0 f=6.007310e-06
x= 1.414216e+00

```

c= 6.007310e-06
 h= 2.7328e-01 hs= 2.7328e-01 mu=1.16e+00 n=6.10e-01 s=7.76e-02 a=4.75e-01

task: b n=0 f=4.287017e-01
 x= 1.414216e+00
 c=-4.287017e-01
 h= 2.2973e-02 hs= 2.2973e-02 mu=0.00e+00 n=2.30e-02 s=2.30e-02 a=1.00e+00

task: b n=1 f=2.297305e-02
 x= 1.565784e+00
 c= 2.297305e-02
 h= 6.5970e-05 hs= 6.5970e-05 mu=0.00e+00 n=5.94e-05 s=5.94e-05 a=1.00e+00

task: b n=2 f=1.109025e-03
 x= 1.558075e+00
 c=-1.109025e-03
 h= 1.5374e-07 hs= 1.5374e-07 mu=0.00e+00 n=1.26e-07 s=1.26e-07 a=1.00e+00

task: b n=3 f=2.610819e-06
 x= 1.558430e+00
 c=-2.610819e-06

task: a n=1 f=1.058663e-01
 x= 2.428708e+00 2.037929e+00
 c=-4.672299e-02 -9.499813e-02
 h= 9.5704e-03 hs= 9.5704e-03 mu=1.57e-01 n=3.10e-02 s=1.44e-03 a=5.67e-01

task: b n=0 f=6.839740e-02
 x= 1.558430e+00
 c=-6.839740e-02
 h= 4.8155e-04 hs= 4.8155e-04 mu=0.00e+00 n=4.82e-04 s=4.82e-04 a=1.00e+00

task: b n=1 f=4.815499e-04
 x= 1.580375e+00
 c= 4.815499e-04

task: a n=2 f=8.482725e-03
 x= 2.497102e+00 2.233778e+00
 c= 7.537533e-03 -3.891300e-03
 h= 2.9346e-05 hs= 2.9346e-05 mu=0.00e+00 n=9.87e-06 s=9.77e-06 a=9.96e-01

task: b n=0 f=2.887597e-03
 x= 1.580375e+00
 c= 2.887597e-03
 h= 8.3463e-07 hs= 8.3463e-07 mu=0.00e+00 n=8.35e-07 s=8.35e-07 a=1.00e+00

task: b n=1 f=8.347711e-07
 x= 1.579461e+00
 c= 8.347711e-07

task: a n=3 f=1.682825e-03
 x= 2.494696e+00 2.235799e+00
 c=-1.681745e-03 6.029810e-05
 h= 7.1070e-07 hs= 7.1070e-07 mu=0.00e+00 n=1.01e-06 s=1.39e-07 a=5.17e-01

task: b n=0 f=8.133102e-04
x= 1.579461e+00
c=-8.133102e-04

task: a n=4 f=1.656567e-04
x= 2.495511e+00 2.236386e+00
c= 1.655548e-04 -5.809680e-06

x=2.50 y=2.24 z=1.58

EXAMPLE FOUR

```

cinit
[/task /a] cnew
{a.c}
  (/x 1 0 10 vary
   /y 2 0 10 vary
   /z 3 0 10 vary
   /x (x-y) cons
   /y (x-z) icons
   ((x-1)*(x-1)+(y-2)*(y-2)+z*exp(z)) mini
  )
while
"\nx=%.2f y=%.2f z=%.2f" [x y z] printf
/all cdel

task a it=1 meq=1 f= 6.0257e+01
x= 1.0000e+00 2.0000e+00 3.0000e+00
c=-1.0000e+00 -2.0000e+00

task a it=3 meq=1 f= 2.0499e+01
x= 2.1731e+00 2.1731e+00 2.1731e+00
c=-5.0266e-06 -1.0053e-05
l= 6.6432e+01

task a it=4 meq=1 f= 1.0251e+01
x= 1.7231e+00 1.7231e+00 1.7231e+00
c=-8.8818e-16 -1.5543e-15
l= 1.3760e+01

task a it=5 meq=1 f= 4.3308e+00
x= 1.4263e+00 1.4263e+00 1.1771e+00
c=-2.2204e-16 2.4912e-01
l= 8.9207e+00

task a it=6 meq=1 f= 1.9493e+00
x= 1.4975e+00 1.4975e+00 7.1147e-01
c= 0.0000e+00 7.8600e-01
l= 3.4323e+00

task a it=7 meq=1 f= 8.3524e-01
x= 1.5129e+00 1.5129e+00 2.5859e-01
c= 0.0000e+00 1.2543e+00
l= 1.6097e+00

task a it=8 meq=1 f= 5.0001e-01
x= 1.4973e+00 1.4973e+00 0.0000e+00
c= 0.0000e+00 1.4973e+00
l= 4.5265e-01

task a it=9 meq=1 f= 5.0000e-01
x= 1.4996e+00 1.4996e+00 0.0000e+00
c= 0.0000e+00 1.4996e+00
l= 4.4555e-03

```



```
task a  it=10  meq=1  f= 5.0000e-01  
x= 1.5000e+00  1.5000e+00  0.0000e+00  
c= 0.0000e+00  1.5000e+00  
l= 9.8042e-04
```

```
x=1.50 y=1.50 z=0.00
```

EXAMPLE FIVE

```

cinit
[/task /a /prt 0] cnew
1.0 1.0 5.0
  {/a.tout exch def
    {a.c}
      {/x 1.0 vary /y 2.0 vary /z 0.0 vary
        /x (-x) diff
        /y (0.5*y) diff
        /z (x-y) diff
      }
    while
    "\ntime=%f x=%f y=%f z=%f" [a.time x y z] printf
  }
for
/all cdel

time=1.00 x=3.680e-01 y=3.298e+00 z=-1.964e+00
time=2.00 x=1.355e-01 y=5.438e+00 z=-6.011e+00
time=3.00 x=4.983e-02 y=8.966e+00 z=-1.298e+01
time=4.00 x=1.834e-02 y=1.478e+01 z=-2.458e+01
time=5.00 x=6.739e-03 y=2.437e+01 z=-4.375e+01

```

EXAMPLE SIX

```

cinit
[/task /a /prt 0] cnew
/interup {"\ntime=%e" [a.time] printf sintrp) def
/trap 1.0 def
/p 1.0 def
1.0 1.0 10.0
  {/a.tout exch def
   {a.c}
    {(a.state<=2 && a.time>=trap) {interup} if
     /x 1.0 vary /y 2.0 vary /z 1.0 vary
     /x (-x*p) diff
     /y (0.5*y) diff
     /z (x-y) diff
    }
   while
   "\ntime=%0.2f x=%0.3e y=%0.3e z=%0.3e" [a.time x y z] printf
  )
for
/all cdel

gps> "ex6.dat" run

time=1.00 x=3.680e-01 y=3.298e+00 z=-9.642e-01
time=1.000000e+00
gps_int> x = y = p =

3.6800e-01
3.2981e+00
1.0000e+00
gps_int> /p 1.1 def

gps_int> /trap 8.2 def resume

time=2.00 x=1.223e-01 y=5.438e+00 z=-5.021e+00
time=3.00 x=4.086e-02 y=8.967e+00 z=-1.200e+01
time=4.00 x=1.352e-02 y=1.479e+01 z=-2.363e+01
time=5.00 x=4.522e-03 y=2.439e+01 z=-4.282e+01
time=6.00 x=1.490e-03 y=4.024e+01 z=-7.452e+01
time=7.00 x=4.985e-04 y=6.636e+01 z=-1.267e+02
time=8.00 x=1.667e-04 y=1.094e+02 z=-2.129e+02
time=8.229812e+00
gps_int> /trap 11.0 def resume

time=9.00 x=5.493e-05 y=1.805e+02 z=-3.551e+02
time=10.00 x=1.838e-05 y=2.977e+02 z=-5.894e+02
gps> quit

```

APPENDIX B

Steady-State Example Four

thermodynamic data for HYDROGEN with flow id = THR-H2
 pc=12.800000, tc=33.200000, tb=20.400000, molwt=2.016000

output of model flows

model	temp	pres	mass	enth	entr	dens	velc	qual
tank_h2	20.0	1.29	7.387	-4.1363e+06	-1.0676e+05	7.716e+01	200.0	0.00
pump_lp	24.8	7.96	7.387	-4.1232e+06	-1.0623e+05	6.783e+01	200.0	0.00
pump_hp	39.0	139.22	7.387	-3.8811e+06	-1.0235e+05	7.148e+01	200.0	1.00
ht_nz	579.4	139.22	7.387	3.9117e+06	-5.8868e+04	5.672e+00	200.0	1.00
sp_2	579.4	139.22	5.171	3.9117e+06	-5.8868e+04	5.672e+00	200.0	1.00
sp_1	579.4	139.22	3.620	3.9117e+06	-5.8868e+04	5.672e+00	200.0	1.00
gt_lp	562.8	85.91	3.620	3.6566e+06	-5.7260e+04	3.658e+00	200.0	1.00
sp_2	579.4	139.22	2.216	3.9117e+06	-5.8868e+04	5.672e+00	200.0	1.00
sp_1	579.4	139.22	1.551	3.9117e+06	-5.8868e+04	5.672e+00	200.0	1.00
gt_hp	521.6	85.74	1.551	3.0768e+06	-5.8324e+04	3.935e+00	200.0	1.00
mix turb	550.4	85.74	5.171	3.4827e+06	-5.7565e+04	3.732e+00	200.0	1.00
mix reac	559.6	85.74	7.387	3.6114e+06	-5.7333e+04	3.672e+00	200.0	1.00
reactor	2930.0	85.74	7.387	4.2500e+07	-3.1131e+04	7.145e-01	200.0	1.00
ht_nz	2498.4	85.74	7.387	3.4708e+07	-3.4007e+04	8.371e-01	200.0	1.00
nozzle	757.1	0.10	7.387	6.3970e+06	-2.5123e+04	3.245e-03	7527.4	1.00

tank_h2 dt=0.0000e+00 dp=0.0000e+00 dm=0.0000e+00 dh=0.0000e+00
 pump_lp eff=6.7000e-01 power=-9.6565e+04
 pump_hp eff=8.1000e-01 power=-1.7881e+06
 ht_nz heat=5.7566e+07 lmt d=2.4046e+03
 sp_2 sr=3.0000e-01
 sp_1 sr=3.0000e-01
 gt_lp eff=2.3000e-01 power=9.2323e+05
 gt_hp eff=7.5000e-01 power=1.2951e+06
 reactor heat=2.8727e+08
 nozzle eff=8.5000e-01 area=3.0240e-01 vel=7.5274e+03 mach=3.5986e+00
 thrust=5.8669e+04 impulse=8.1042e+02

output of model powers

model	input	loss	prod	cons
pump_lp	0.0000e+00	0.0000e+00	0.0000e+00	9.6565e+04
pump_hp	0.0000e+00	0.0000e+00	0.0000e+00	1.7881e+06
gt_lp	0.0000e+00	0.0000e+00	9.2323e+05	0.0000e+00
gt_hp	0.0000e+00	0.0000e+00	1.2951e+06	0.0000e+00
reactor	2.8727e+08	0.0000e+00	0.0000e+00	0.0000e+00

APPENDIX C

Steady-State Example Four with Constraints

thermodynamic data for HYDROGEN with flow id = THR-tH2
 pc=12.800000, tc=33.200000, tb=20.400000, molwt=2.016000

task: a n=0 f=9.624907e+05
 x= 3.000000e-01 3.000000e-01
 c= 8.266618e+05 -4.929691e+05
 h= 4.0590e-01 hs= 4.0590e-01 mu=0.00e+00 n=2.01e-01 s=1.75e-01 a=9.65e-01

task: a n=1 f=6.200406e+05
 x= 6.588144e-01 5.679687e-01
 c= 1.811617e+05 -5.929846e+05
 h= 3.7735e-02 hs= 3.7735e-02 mu=0.00e+00 n=2.83e-02 s=2.81e-02 a=9.99e-01

task: a n=2 f=2.498952e+05
 x= 6.588144e-01 7.362034e-01
 c= 7.301368e+04 -2.389908e+05
 h= 6.1294e-03 hs= 6.1294e-03 mu=0.00e+00 n=1.29e-02 s=1.25e-02 a=9.97e-01

task: a n=3 f=7.841918e-05
 x= 6.588144e-01 8.497832e-01
 c=-2.291224e-05 7.499731e-05

output of model flows

model	temp	pres	mass	enth	entr	dens	velc	qual
tank_h2	20.0	1.29	7.387	-4.1363e+06	-1.0676e+05	7.716e+01	200.0	0.00
pump_lp	24.8	7.96	7.387	-4.1232e+06	-1.0623e+05	6.783e+01	200.0	0.00
pump_hp	39.0	139.22	7.387	-3.8811e+06	-1.0235e+05	7.148e+01	200.0	1.00
ht_nz	579.4	139.22	7.387	3.9117e+06	-5.8868e+04	5.672e+00	200.0	1.00
sp_2	579.4	139.22	2.520	3.9117e+06	-5.8868e+04	5.672e+00	200.0	1.00
sp_1	579.4	139.22	0.379	3.9117e+06	-5.8868e+04	5.672e+00	200.0	1.00
gt_lp	562.8	85.91	0.379	3.6566e+06	-5.7260e+04	3.658e+00	200.0	1.00
sp_2	579.4	139.22	4.867	3.9117e+06	-5.8868e+04	5.672e+00	200.0	1.00
sp_1	579.4	139.22	2.142	3.9117e+06	-5.8868e+04	5.672e+00	200.0	1.00
gt_hp	521.6	85.74	2.142	3.0768e+06	-5.8324e+04	3.935e+00	200.0	1.00
mix_turb	527.8	85.74	2.520	3.1639e+06	-5.8157e+04	3.889e+00	200.0	1.00
mix_reac	562.8	85.74	7.387	3.6566e+06	-5.7252e+04	3.651e+00	200.0	1.00
reactor	2930.0	85.74	7.387	4.2500e+07	-3.1131e+04	7.145e-01	200.0	1.00
ht_nz	2498.4	85.74	7.387	3.4708e+07	-3.4007e+04	8.371e-01	200.0	1.00
nozzle	757.1	0.10	7.387	6.3970e+06	-2.5123e+04	3.245e-03	7527.4	1.00

tank_h2 dt=0.0000e+00 dp=0.0000e+00 dm=0.0000e+00 dh=0.0000e+00
 pump_lp eff=6.7000e-01 power=-9.6565e+04
 pump_hp eff=8.1000e-01 power=-1.7881e+06
 ht_nz heat=5.7566e+07 lmtcd=2.4046e+03

sp_2 sr=6.5881e-01
 sp_1 sr=8.4978e-01
 gt_lp cff=2.3000e-01 power=9.6565e+04
 gt_hp cff=7.5000e-01 power=1.7881e+06
 reactor heat=2.8694e+08
 nozzle cff=8.5000e-01 area=3.0240e-01 vel=7.5274e+03 mach=3.5986e+00
 thrust=5.8669e+04 impulse=8.1042e+02

output of model powers

model	input	loss	prod	cons
pump_lp	0.0000e+00	0.0000e+00	0.0000e+00	9.6565e+04
pump_hp	0.0000e+00	0.0000e+00	0.0000e+00	1.7881e+06
gt_lp	0.0000e+00	0.0000e+00	9.6565e+04	0.0000e+00
gt_hp	0.0000e+00	0.0000e+00	1.7881e+06	0.0000e+00
reactor	2.8694e+08	0.0000e+00	0.0000e+00	0.0000e+00

APPENDIX D

Dynamic Example One at Design Point

thermodynamic data for HYDROGEN with flow id = THR-tH2
 pc=12.800000, tc=33.200000, tb=20.400000, molwt=2.016000

task: sta n=0 f=7.563290e+04
 x= 5.276954e-02 5.000000e-02 2.000000e-02
 c= 6.425210e-01 3.602149e+01 -7.563289e+04
 h= 3.7461e-04 hs= 3.7461e-04 mu=1.01e+00 n=7.54e-04 s=1.02e-04 a=7.46e-01

task: sta n=1 f=1.973103e+04
 x= 4.488375e-02 7.153655e-02 3.305220e-02
 c= 1.428151e-01 1.134402e+02 -1.973071e+04
 h= 3.7498e-05 hs= 3.7498e-05 mu=8.88e-03 n=8.54e-05 s=5.45e-05 a=9.76e-01

task: sta n=2 f=1.175786e+03
 x= 4.153330e-02 8.905709e-02 3.750814e-02
 c=-3.152815e-03 4.201986e+01 -1.175035e+03
 h= 1.8831e-06 hs= 1.8831e-06 mu=0.00e+00 n=3.27e-06 s=5.28e-07 a=6.27e-01

task: sta n=3 f=9.639517e+01
 x= 4.117667e-02 9.321568e-02 3.777773e-02
 c=-1.281143e-03 1.520944e+01 -9.518771e+01
 h= 2.3726e-07 hs= 2.3726e-07 mu=0.00e+00 n=6.48e-07 s=6.97e-08 a=5.00e-01

task: sta n=4 f=3.650815e+00
 x= 4.102025e-02 9.509075e-02 3.780441e-02
 c=-1.123252e-04 2.114729e+00 -2.975966e+00
 h= 4.5750e-09 hs= 4.5750e-09 mu=0.00e+00 n=1.56e-08 s=1.36e-09 a=4.52e-01

task: sta n=5 f=1.629713e-01
 x= 4.099520e-02 9.538108e-02 3.780582e-02
 c=-4.213630e-06 7.386692e-02 -1.452699e-01
 h= 5.5828e-12 hs= 5.5828e-12 mu=0.00e+00 n=2.06e-11 s=1.64e-12 a=4.38e-01

task: sta n=6 f=3.698913e-03
 x= 4.099430e-02 9.539164e-02 3.780588e-02
 c=-7.882591e-09 -5.644356e-06 3.698909e-03
 h= 7.9740e-19 hs= 7.9740e-19 mu=0.00e+00 n=5.52e-18 s=1.24e-19 a=5.11e-01

convergence of independent variables in task sta

output of model flows

model	temp	pres	mass	enth	entr	dens	velc	qual
gas_h2	20.0	3.00	1.000	-4.1681e+06	-1.0917e+05	7.739e+01	6.6	0.00
valv_tsov	20.0	2.94	1.000	-4.1681e+06	-1.0915e+05	7.745e+01	6.6	0.00
pi_l	20.0	2.94	1.000	-4.1681e+06	-1.0915e+05	7.745e+01	6.6	0.00

pump_tp	33.4	86.94	1.000	-3.9990e+06	-1.0425e+05	7.059e+01	6.6	1.00
valv_psov	33.4	85.20	1.000	-3.9990e+06	-1.0416e+05	7.024e+01	6.6	1.00
pi_2	33.4	85.19	1.000	-3.9990e+06	-1.0415e+05	7.023e+01	7.3	1.00
ht_r	80.0	85.02	1.000	-3.1339e+06	-8.5652e+04	2.814e+01	4.5	1.00
sp_1	80.0	85.02	0.905	-3.1339e+06	-8.5652e+04	2.814e+01	16.4	1.00
pi_3	80.0	85.02	0.905	-3.1339e+06	-8.5651e+04	2.814e+01	45.5	1.00
reac_1	3000.0	81.54	0.905	4.3783e+07	-3.0490e+04	6.639e-01	693.9	1.00
sp_2	3000.0	81.54	0.870	4.3783e+07	-3.0490e+04	6.639e-01	667.7	1.00
exnz_1	1142.1	1.10	0.870	1.2123e+07	-2.8902e+04	2.631e-02	8318.3	1.00
sp_1	80.0	85.02	0.095	-3.1339e+06	-8.5652e+04	2.814e+01	43.2	1.00
valv_tev	79.6	81.54	0.095	-3.1339e+06	-8.5496e+04	2.728e+01	43.2	1.00
pi_5	79.6	81.53	0.095	-3.1339e+06	-8.5496e+04	2.728e+01	44.5	1.00
sp_2	3000.0	81.54	0.034	4.3783e+07	-3.0490e+04	6.639e-01	26.2	1.00
pi_4	3000.0	81.53	0.034	4.3783e+07	-3.0489e+04	6.638e-01	26.2	1.00
mx_1	948.5	81.53	0.130	9.2476e+06	-4.9486e+04	2.078e+00	198.5	1.00
pi_6	948.5	81.52	0.130	9.2476e+06	-4.9485e+04	2.078e+00	198.5	1.00
valv_scv	950.0	40.76	0.130	9.2476e+06	-4.6593e+04	1.046e+00	198.5	1.00
pi_7	950.0	40.76	0.130	9.2476e+06	-4.6593e+04	1.046e+00	394.5	1.00
gt_tp	862.3	27.17	0.130	7.9430e+06	-4.6350e+04	7.697e-01	394.5	1.00
exnz_tr	725.5	14.71	0.130	5.9485e+06	-4.6326e+04	4.963e-01	2046.4	1.00

output of model parameters

```

gas_h2      diam=5.0000e-02 area=1.9635e-03
valv_tsov   dp=6.0000e-02 pf=2.0000e-02
pi_1         dp=5.8800e-04
              length=1.0000e+00 diam=5.0000e-02 area=1.9635e-03
              rat_pf=2.0000e-04 rat_n=1.0000e+00
              num_par=1.00
pump_ip      rpm=6.0000e+04 dp=8.4001e+01 eff=6.5000e-01 power=-1.6907e+05
              rat_dp=8.4000e+01 rat_torque=2.0577e+01 rat_rpm=6.0000e+04
              rat_m=1.0000e+00 rat_eff=8.5000e-01 inertia=1.0000e-01
valv_psov    dp=1.7388e+00 pf=2.0000e-02
pi_2         dp=8.5201e-03
              length=1.0000e+00 diam=5.0000e-02 area=1.9635e-03
              rat_pf=1.0000e-04 rat_n=1.0000e+00
              num_par=1.00
ht_r         heat=8.6509e+05 dp=1.7039e-01
              ua=1.0000e+05 cpwall=1.0000e+03 mwall=0.0000e+00 vol=7.853982e-03
              diam=1.0000e-01 area=7.8540e-03 length=1.0000e+00
              tconst=1.0000e+01
sp_1         sr=9.5392e-02
              diam0=5.0000e-02 diam1=1.0000e-02 area0=1.9635e-03 area1=7.8540e-05
pi_3         dp=6.9575e-03
              length=1.0000e+00 diam=3.0000e-02 area=7.0686e-04
              rat_pf=1.0000e-04 rat_m=1.0000e+00
              num_par=1.00
reac_1       power=4.2442e+07 tfuel=3.0054e+03 tclad=0.0000e+00 dp=3.4785e+00
              lmtcd=4.6438e+02
              mfuel=2.2500e+02 mclad=0.0000e+00 cpfuel=1.0000e+03 cpcld=1.0000e+03
              uclad=1.3000e+05 ucool=1.0000e+05
              vol=1.0000e+00 diam=5.0000e-02 area=1.9635e-03
sp_2         sr=3.7806e-02
              diam0=5.0000e-02 diam1=5.0000e-02 area0=1.9635e-03 area1=1.9635e-03
pi_4         dp=8.1537e-03

```



```

length=1.0000e+00 diam=5.0000e-02 area=1.9635e-03
rat_pf=1.0000e-04 rat_m=3.0000e-02
num_par=1.00
exnz_l mreq=8.7041e-01 thrust=8.9090e+03 impulse=1.0444e+03
diam=2.6459e-02 area=5.4985e-04
mach=3.3662e+00 aexp=1.5000e-02
valv_tcv dp=3.4854e+00 pf=4.0994e-02
pi_5 dp=8.1537e-03
length=1.0000e+00 diam=1.0000e-02 area=7.8540e-05
rat_pf=1.0000e-04 rat_m=8.0000e-02
num_par=1.00
mx_l diam=2.0000e-02 area=3.1416e-04
pi_6 dp=8.1529e-03
length=1.0000e+00 diam=2.0000e-02 area=3.1416e-04
rat_pf=1.0000e-04 rat_m=1.1000e-01
num_par=1.00
valv_scv dp=4.0760e+01 pf=5.0000e-01
pi_7 dp=4.0760e-03
length=1.0000e+00 diam=2.0000e-02 area=3.1416e-04
rat_pf=1.0000e-04 rat_m=1.1000e-01
num_par=1.00
gt_tp rpm=6.0000e+04 eff=8.6282e-01 power=1.6907e+05
cmass=1.0000e+00 cspeed=1.0000e+00
rat_cmass=9.8004e-02 rat_cspeed=1.9467e+03
rat_pr=1.5000e+00 inertia=2.0000e-01
exnz_tp mreq=1.2959e-01 thrust=4.5577e+02 impulse=3.5888e+02
diam=1.2760e-02 area=1.2788e-04
shft_l rpm=6.0000e+04 power=3.6989e-03 inertia=3.0000e-01

```

APPENDIX E

Dynamic Example One

thermodynamic data for HYDROGEN with flow id = THR-tH2
 pc=12.800000, tc=33.200000, tb=20.400000, molwt=2.016000

***** time = 0.0000e+00 *****

output of model flows

model	temp	pres	mass	enth	entr	dens	velc	qual
gas_h2	20.0	3.00	0.206	-4.1681e+06	-1.0917e+05	7.739e+01	1.4	0.00
valv_tsov	20.0	2.94	0.206	-4.1681e+06	-1.0915e+05	7.745e+01	1.4	0.00
pi_1	20.0	2.94	0.206	-4.1681e+06	-1.0915e+05	7.745e+01	1.4	0.00
pump_tp	20.0	2.94	0.206	-4.1681e+06	-1.0915e+05	7.745e+01	1.4	0.00
valv_psov	19.9	2.88	0.206	-4.1681e+06	-1.0914e+05	7.750e+01	1.4	0.00
pi_2	19.9	2.88	0.206	-4.1681e+06	-1.0914e+05	7.750e+01	1.4	0.00
ht_r	80.0	2.88	0.206	-2.8889e+06	-6.9291e+04	8.922e-01	29.4	1.00
sp_1	80.0	2.88	0.191	-2.8889e+06	-6.9291e+04	8.922e-01	109.2	1.00
pi_3	80.0	2.88	0.191	-2.8889e+06	-6.9291e+04	8.922e-01	303.4	1.00
reac_1	100.0	2.88	0.191	-2.6266e+06	-6.6350e+04	7.094e-01	137.4	1.00
sp_2	100.0	2.88	0.176	-2.6266e+06	-6.6350e+04	7.094e-01	126.6	1.00
exnz_1	24.0	0.03	0.176	-3.6008e+06	-6.5922e+04	3.097e-02	1417.4	1.00
sp_1	80.0	2.88	0.015	-2.8889e+06	-6.9291e+04	8.922e-01	212.4	1.00
valv_tcv	80.0	2.88	0.015	-2.8889e+06	-6.9283e+04	8.905e-01	212.4	1.00
pi_5	80.0	2.88	0.015	-2.8889e+06	-6.9283e+04	8.905e-01	212.8	1.00
sp_2	100.0	2.88	0.015	-2.6266e+06	-6.6350e+04	7.094e-01	10.7	1.00
pi_4	100.0	2.88	0.015	-2.6266e+06	-6.6350e+04	7.093e-01	10.7	1.00
mx_1	90.0	2.88	0.030	-2.7575e+06	-6.7732e+04	7.894e-01	120.3	1.00
pi_6	90.0	2.88	0.030	-2.7575e+06	-6.7732e+04	7.894e-01	120.3	1.00
valv_scv	90.0	2.82	0.030	-2.7575e+06	-6.7649e+04	7.737e-01	120.3	1.00
pi_7	90.0	2.82	0.030	-2.7575e+06	-6.7649e+04	7.737e-01	122.7	1.00
gt_tp	86.1	1.94	0.030	-2.8056e+06	-6.6657e+04	5.550e-01	122.7	1.00
exnz_tp	70.5	1.03	0.030	-3.0052e+06	-6.6609e+04	3.589e-01	649.7	1.00

output of model parameters

gas_h2	diam=5.0000e-02 area=1.9635e-03
valv_tsov	dp=6.0000e-02 pf=2.0000e-02
pi_1	dp=2.5000e-05
	length=1.0000e+00 diam=5.0000e-02 area=1.9635e-03
	rat_pf=2.0000e-04 rat_m=1.0000e+00
	num_par=1.00
pump_tp	rpm=5.0000e+03 dp=0.0000e+00 eff=6.5000e-01 power=0.0000e+00
	rat_dp=8.4000e+01 rat_torque=2.0577e+01 rat_rpm=6.0000e+04
	rat_m=1.0000e+00 rat_eff=8.5000e-01 inertia=1.0000e-02
valv_psov	dp=5.8800e-02 pf=2.0000e-02

```

pi_2      dp=1.2250e-05
          length=1.0000e+00 diam=5.0000e-02 area=1.9635e-03
          rat_pf=1.0000e-04 rat_m=1.0000e+00
          num_par=1.00
ht_r      heat=2.6377e+05 dp =2.4499e-04
          ua=1.0000e+05 cpwall=1.0000e+03 mwall=0.0000e+00 vol=7.853982e-03
          diam=1.0000e-01 area=7.8540e-03 length=1.0000e+00
          tconst=1.0000e+01
sp_1      sr=7.2185e-02
          diam0=5.0000e-02 diam1=1.0000e-02 area0=1.9635e-03 areal=7.8540e-05
pi_3      dp=1.0544e-05
          length=1.0000e+00 diam=3.0000e-02 area=7.0686e-04
          rat_pf=1.0000e-04 rat_m=1.0000e+00
          num_par=1.00
reac_1    power=5.0186e+04 tfuel=3.0000e+02 tclad=0.0000e+00 dp=5.2720e-03
          lmtcd=2.0984e+02
          mfuel=2.0000e+02 mclad=0.0000e+00 cpfuel=1.0000e+03 cpcld=1.0000e+03
          uclad=1.3000e+06 ucool=1.0000e+04
          vol=1.0000e+00 diam=5.0000e-02 area=1.9635e-03
sp_2      sr=7.8085e-02
          diam0=5.0000e-02 diam1=5.0000e-02 area0=1.9635e-03 areal=1.9635e-03
pi_4      dp=7.1304e-05
          length=1.0000e+00 diam=5.0000e-02 area=1.9635e-03
          rat_pf=1.0000e-04 rat_m=3.0000e-02
          num_par=1.00
exnz_1    mreq=1.7637e-01 thrust=2.9586e+02 impulse=1.7117e+02
          diam=2.6459e-02 area=5.4984e-04
          mach=3.7422e+00 aexp=1.5000e-02
valv_tcv  dp=5.3439e-03 pf=1.8549e-03
pi_5      dp=9.9541e-06
          length=1.0000e+00 diam=1.0000e-02 area=7.8540e-05
          rat_pf=1.0000e-04 rat_m=8.0000e-02
          num_par=1.00
mx_1      diam=2.0000e-02 area=3.1416e-04
pi_6      dp=2.1137e-05
          length=1.0000e+00 diam=2.0000e-02 area=3.1416e-04
          rat_pf=1.0000e-04 rat_m=1.1000e-01
          num_par=1.00
valv_scv  dp=5.7511e-02 pf=2.0000e-02
pi_7      dp=2.0714e-05
          length=1.0000e+00 diam=2.0000e-02 area=3.1416e-04
          rat_pf=1.0000e-04 rat_m=1.1000e-01
          num_par=1.00
gt_tp     rpm=5.0000e+03 eff=3.6967e-01 power=1.4364e+03
          cmass=1.0244e+00 cspeed=2.7074e-01
          rat_cmass=9.8004e-02 rat_cspeed=1.9467e+03
          rat_pr=1.5000e+00 inertia=2.0000e-02
exnz_tp   mreq=2.9826e-02 thrust=3.2667e+01 impulse=1.1177e+02
          diam=1.2760e-02 area=1.2788e-04
shft_1    rpm=5.0000e+03 power=1.4364e+03 inertia=3.0000e-02
cntl_1    cntl=2.0000e-02 err=-2.7500e+00 icrr=0.0000e+00 derr=0.0000e+00
          k=1.0000e+00 tp=1.0000e+00 ti=0.0000e+00 td=1.0000e+00

```

***** time = 1.0000e+01 *****

output of model flows

model	temp	pres	mass	enth	entr	dens	velc	qual
gas_h2	20.0	3.00	0.273	-4.1681e+06	-1.0917e+05	7.739e+01	1.8	0.00
valv_tsov	20.0	2.94	0.273	-4.1681e+06	-1.0915e+05	7.745e+01	1.8	0.00
pi_1	20.0	2.94	0.273	-4.1681e+06	-1.0915e+05	7.745e+01	1.8	0.00
pump_tp	23.4	8.15	0.273	-4.1576e+06	-1.0812e+05	7.151e+01	1.8	0.00
valv_psov	23.3	7.98	0.273	-4.1576e+06	-1.0813e+05	7.162e+01	1.8	0.00
pi_2	23.3	7.98	0.273	-4.1576e+06	-1.0813e+05	7.162e+01	1.9	0.00
ht_r	59.3	7.98	0.273	-3.1908e+06	-7.7807e+04	3.493e+00	9.9	1.00
sp_1	59.3	7.98	0.268	-3.1908e+06	-7.7807e+04	3.493e+00	39.1	1.00
pi_3	59.3	7.98	0.268	-3.1908e+06	-7.7807e+04	3.493e+00	108.6	1.00
reac_1	428.5	7.95	0.268	1.7699e+06	-5.1176e+04	4.548e-01	300.3	1.00
sp_2	428.5	7.95	0.233	1.7699e+06	-5.1176e+04	4.548e-01	261.3	1.00
exnz_1	110.2	0.09	0.233	2.4861e+06	-5.0583e+04	1.933e-02	2971.1	1.00
sp_1	59.3	7.98	0.005	-3.1908e+06	-7.7807e+04	3.493e+00	17.0	1.00
valv_tcv	59.3	7.95	0.005	-3.1908e+06	-7.7792e+04	3.480e+00	17.0	1.00
pi_5	59.3	7.95	0.005	-3.1908e+06	-7.7792e+04	3.480e+00	17.1	1.00
sp_2	428.5	7.95	0.035	1.7699e+06	-5.1176e+04	4.548e-01	39.0	1.00
pi_4	428.5	7.95	0.035	1.7699e+06	-5.1176e+04	4.548e-01	39.0	1.00
mx_1	385.7	7.95	0.039	1.1833e+06	-5.2619e+04	5.053e-01	248.5	1.00
pi_6	385.7	7.95	0.039	1.1833e+06	-5.2619e+04	5.053e-01	248.5	1.00
valv_scv	385.7	7.79	0.039	1.1833e+06	-5.2535e+04	4.952e-01	248.5	1.00
pi_7	385.7	7.79	0.039	1.1833e+06	-5.2535e+04	4.952e-01	253.6	1.00
gt_tp	368.2	5.34	0.039	9.4542e+05	-5.1605e+04	3.557e-01	253.6	1.00
exnz_tp	304.3	2.85	0.039	8.1564e+04	-5.1587e+04	2.298e-01	1343.4	1.00

output of model parameters

gas_h2	diam=5.0000e-02 area=1.9635e-03
valv_tsov	dp=6.0000e-02 pf=2.0000e-02
pi_1	dp=4.3764e-05
pump_tp	rpm=1.5073e+04 dp=5.2063e+00 eff=5.5000e-01 power=-2.8588e+03
valv_psov	dp=1.6293e-01 pf=2.0000e-02
pi_2	dp=5.9419e-05
ht_r	heat=2.6377e+05 dp=1.1884e-03
sp_1	sr=1.7098e-02
pi_3	dp=5.7396e-05
reac_1	power=2.8345e+07 tfuel=9.5966e+02 tclad=0.0000e+00 dp=2.8698e-02 lmttd=6.9961e+02
sp_2	sr=1.2972e-01
pi_4	dp=7.9533e-04
exnz_1	mreq=2.3337e-01 thrust=8.2509e+02 impulse=3.6077e+02
valv_tcv	dp=2.9548e-02 pf=3.7017e-03
pi_5	dp=2.7039e-06
pi_6	dp=1.0228e-04
valv_scv	dp=1.5905e-01 pf=2.0000e-02
pi_7	dp=1.0024e-04
gt_tp	rpm=1.5073e+04 eff=4.1819e-01 power=9.3860e+03 cmass=1.0143e+00 cspeed=3.9427e-01
exnz_tp	mreq=3.9449e-02 thrust=8.9914e+01 impulse=2.3258e+02
shft_1	rpm=1.5073e+04 power=6.5272e+03 inertia=3.0000e-02
cntl_1	cntl=2.0000e-02 err=-2.2464e+00 ierr=0.0000e+00 derr=0.0000e+00

***** time = 2.0000e+01 *****

output of model flows

model	temp	pres	mass	enth	entr	dens	velc	qual
gas_h2	20.0	3.00	0.712	-4.1681e+06	-1.0917e+05	7.739e+01	4.7	0.00
valv_tsov	20.0	2.94	0.712	-4.1681e+06	-1.0915e+05	7.745e+01	4.7	0.00
pi_1	20.0	2.94	0.712	-4.1681e+06	-1.0915e+05	7.745e+01	4.7	0.00
pump_tp	27.7	30.07	0.712	-4.1135e+06	-1.0629e+05	6.797e+01	4.7	1.00
valv_psov	27.7	29.47	0.712	-4.1135e+06	-1.0626e+05	6.784e+01	4.7	1.00
pi_2	27.7	29.47	0.712	-4.1135e+06	-1.0626e+05	6.784e+01	5.3	1.00
ht_r	41.1	29.44	0.712	-3.7431e+06	-9.4009e+04	3.474e+01	2.6	1.00
sp_1	41.1	29.44	0.701	-3.7431e+06	-9.4009e+04	3.474e+01	10.3	1.00
pi_3	41.1	29.44	0.701	-3.7431e+06	-9.4009e+04	3.474e+01	28.6	1.00
reac_1	803.7	28.71	0.701	7.0864e+06	-4.7608e+04	8.721e-01	409.4	1.00
sp_2	803.7	28.71	0.609	7.0864e+06	-4.7608e+04	8.721e-01	355.7	1.00
exnz_1	225.7	0.33	0.609	-9.6784e+05	-4.6659e+04	3.573e-02	4125.4	1.00
sp_1	41.1	29.44	0.011	-3.7431e+06	-9.4009e+04	3.474e+01	4.1	1.00
valv_tcv	40.9	28.71	0.011	-3.7431e+06	-9.3968e+04	3.433e+01	4.1	1.00
pi_5	40.9	28.71	0.011	-3.7431e+06	-9.3968e+04	3.433e+01	4.1	1.00
sp_2	803.7	28.71	0.092	7.0864e+06	-4.7608e+04	8.721e-01	53.7	1.00
pi_4	803.7	28.71	0.092	7.0864e+06	-4.7608e+04	8.720e-01	53.7	1.00
mx_1	723.3	28.71	0.103	5.9223e+06	-4.9134e+04	9.684e-01	338.8	1.00
pi_6	723.3	28.71	0.103	5.9223e+06	-4.9134e+04	9.684e-01	338.8	1.00
valv_scv	723.3	28.13	0.103	5.9223e+06	-4.9050e+04	9.491e-01	338.8	1.00
pi_7	723.3	28.13	0.103	5.9223e+06	-4.9049e+04	9.490e-01	345.7	1.00
gt_tp	673.7	19.05	0.103	5.2070e+06	-4.8457e+04	6.913e-01	345.7	1.00
exnz_tp	563.1	10.26	0.103	3.6394e+06	-4.8435e+04	4.461e-01	1812.3	1.00

output of model parameters

gas_h2	diam=5.0000e-02 area=1.9635e-03
valv_tsov	dp=6.0000e-02 pf=2.0000e-02
pi_1	dp=2.9826e-04
pump_tp	rpm=3.5812e+04 dp=2.7130e+01 eff=6.5000e-01 power=-3.8891e+04
valv_psov	dp=6.0140e-01 pf=2.0000e-02
pi_2	dp=1.4948e-03
ht_r	heat=2.6377e+05 dp =2.9894e-02
sp_1	sr=1.5556e-02
pi_3	dp=1.4471e-03
reac_1	power=4.2109e+07 tfuel=2.5196e+03 tclad=0.0000e+00 dp=7.2350e-01 lmt=2.0739e+03
sp_2	sr=1.3120e-01
pi_4	dp=2.8712e-03
exnz_1	mreq=6.0903e-01 thrust=3.0118e+03 impulse=5.0452e+02
valv_tcv	dp=7.2764e-01 pf=2.4718e-02
pi_5	dp=5.5062e-05
pi_6	dp=2.5203e-03
valv_scv	dp=5.7414e-01 pf=2.0000e-02
pi_7	dp=2.4697e-03
gt_tp	rpm=3.5812e+04 eff=6.4662e-01 power=7.3725e+04 cmass=1.0054e+00 cspeed=6.8401e-01
exnz_tp	mreq=1.0319e-01 thrust=3.1967e+02 impulse=3.1649e+02
shft_1	rpm=3.5812e+04 power=3.4834e+04 inertia=3.0000e-02

cntl_1 cntl=2.0000e-02 err=-1.2094e+00 ierr=0.0000e+00 derr=0.0000e+00

***** time = 3.0000e+01 *****

output of model flows

model	temp	pres	mass	enth	entr	dens	velc	qual
gas_h2	20.0	3.00	1.211	-4.1681e+06	-1.0917e+05	7.739e+01	8.0	0.00
valv_tsov	20.0	2.94	1.211	-4.1681e+06	-1.0915e+05	7.745e+01	8.0	0.00
pi_1	20.0	2.94	1.211	-4.1681e+06	-1.0915e+05	7.745e+01	8.0	0.00
pump_tp	31.3	65.56	1.211	-4.0421e+06	-1.0493e+05	6.969e+01	8.0	1.00
valv_psov	31.3	64.25	1.211	-4.0421e+06	-1.0487e+05	6.941e+01	8.0	1.00
pi_2	31.3	64.24	1.211	-4.0421e+06	-1.0487e+05	6.941e+01	8.9	1.00
ht_r	42.3	64.05	1.211	-3.8242e+06	-9.7708e+04	5.372e+01	2.9	1.00
sp_1	42.3	64.05	1.191	-3.8242e+06	-9.7708e+04	5.372e+01	11.3	1.00
pi_3	42.3	64.05	1.191	-3.8242e+06	-9.7707e+04	5.372e+01	31.4	1.00
reac_1	1118.0	60.84	1.191	1.1791e+07	-4.5795e+04	1.323e+00	458.2	1.00
sp_2	1118.0	60.84	1.086	1.1791e+07	-4.5795e+04	1.323e+00	417.9	1.00
exnz_1	334.1	0.72	1.086	4.8451e+05	-4.4643e+04	5.287e-02	4917.5	1.00
sp_1	42.3	64.05	0.020	-3.8242e+06	-9.7708e+04	5.372e+01	4.8	1.00
valv_tcv	42.1	60.84	0.020	-3.8242e+06	-9.7583e+04	5.289e+01	4.8	1.00
pi_5	42.1	60.84	0.020	-3.8242e+06	-9.7583e+04	5.289e+01	4.9	1.00
sp_2	1118.0	60.84	0.105	1.1791e+07	-4.5795e+04	1.323e+00	40.3	1.00
pi_4	1118.0	60.84	0.105	1.1791e+07	-4.5795e+04	1.323e+00	40.3	1.00
mx_1	949.2	60.84	0.125	9.2469e+06	-4.8262e+04	1.556e+00	256.2	1.00
pi_6	949.2	60.83	0.125	9.2469e+06	-4.8262e+04	1.556e+00	256.2	1.00
valv_scv	950.0	39.38	0.125	9.2469e+06	-4.6451e+04	1.011e+00	256.2	1.00
pi_7	950.0	39.38	0.125	9.2469e+06	-4.6451e+04	1.011e+00	394.4	1.00
gt_tp	866.2	26.32	0.125	8.0004e+06	-4.6151e+04	7.423e-01	394.4	1.00
exnz_tp	728.9	14.25	0.125	5.9974e+06	-4.6126e+04	4.785e-01	2050.7	1.00

output of model parameters

gas_h2	diam=5.0000e-02 area=1.9635e-03
valv_tsov	dp=6.0000e-02 pf=2.0000e-02
pi_1	dp=5.8800e-04
pump_tp	rpm=5.7052e+04 dp=6.2619e+01 eff=6.5000e-01 power=-1.5262e+05
valv_psov	dp=1.3112e+00 pf=2.0000e-02
pi_2	dp=6.4247e-03
ht_r	heat=2.6377e+05 dp =1.8840e-01
sp_1	sr=1.6848e-02
pi_3	dp=6.4052e-03
reac_1	power=4.2109e+07 tfuel=3.3436e+03 tclad=0.0000e+00 dp=3.2023e+00 lmttd=2.7282e+03
sp_2	sr=8.8040e-02
pi_4	dp=6.0843e-03
exnz_1	mreq=1.0857e+00 thrust=6.4321e+03 impulse=6.0452e+02
valv_tcv	dp=3.2144e+00 pf=5.0184e-02
pi_5	dp=3.9568e-04
pi_6	dp=6.0837e-03
valv_scv	dp=2.1448e+01 pf=3.5258e-01
pi_7	dp=3.9383e-03
gt_tp	rpm=5.7052e+04 eff=8.2917e-01 power=1.5608e+05

exnz_tp cmass=1.0000e+00 cspeed=9.5084e-01
shft_l mreq=1.2522e-01 thrust=4.4137e+02 impulse=3.5968e+02
cntl_l rpm=5.7052e+04 power=3.4589e+03 inertia=3.0000e-02
cntl_l cntl=3.5258e-01 err=-1.4742e-01 ierr=0.0000e+00 derr=0.0000e+00

APPENDIX F

Thermionic System Example

task: a n=0 f=2.244738e+04
 x= 8.000000e+05
 c= 2.244738e+04
 h= 2.3043e+11 hs= 2.3043e+11 mu=0.00e+00 n=2.30e+11 s=2.30e+11 a=1.00e+00 b=0 b=0

task: a n=1 f=8.971750e+03
 x= 5.599855e+05
 c= 8.971750e+03
 h= 3.6809e+10 hs= 3.6809e+10 mu=2.00e+00 n=2.55e+10 s=2.55e+10 a=1.00e+00

task: a n=2 f=5.562803e+02
 x= 4.401386e+05
 c= 5.562803e+02
 h= 1.4151e+08 hs= 1.4151e+08 mu=1.00e+00 n=6.28e+07 s=6.28e+07 a=1.00e+00

task: a n=3 f=3.960431e+01
 x= 4.322164e+05
 c= -3.960431e+01
 h= 7.1728e+05 hs= 7.1728e+05 mu=0.00e+00 n=2.77e+05 s=2.77e+05 a=1.00e+00

task: a n=4 f=1.523882e-01
 x= 4.327430e+05
 c= 1.523882e-01
 h= 1.0620e+01 hs= 1.0620e+01 mu=0.00e+00 n=4.07e+00 s=4.07e+00 a=1.00e+00

task: a n=5 f=4.142651e-05
 x= 4.327409e+05
 c= 4.142651e-05

output of model flows

model	power watts	voltage volts	current amps/s
reac_l	5.626e+04	0.000	0.000e+00
ti_l	5.626e+04	6.000	9.376e+03
sp_shunt	5.569e+04	6.000	9.282e+03
res_ti	4.708e+04	5.072	9.282e+03
bc_l	4.002e+04	100.000	4.002e+02
res_bc	4.000e+04	99.960	4.002e+02
bus_l	4.000e+04	99.960	4.002e+02
reac_l.s	3.765e+05	0.000	0.000e+00
rad_prim	3.765e+05	0.000	0.000e+00
sp_shunt.s	5.626e+02	6.000	9.376e+01
rad_shunt	5.626e+02	0.000	0.000e+00
bc_l.s	7.062e+03	0.000	0.000e+00
rad_bc	7.062e+03	0.000	0.000e+00

output of model parameters

```

reac_l      pow=432740.9382 eff=0.1300 radius=0.20 height=158.29
            radiusrs=88.670 volrs=0.000 heightrs=0.370 sep=10.000
ti_l        v=6.0000 vconv=0.67 iconv=62.00 ncs=3.96 ncp=151.23
sp_shunt    sr=0.0100
bc_l        eff=0.8500 v=100.0000
res_ti      r=0.0001
res_bc      r=0.0001
rad_prim    area=78.4625 t=1000.00 tspace=255.00 e=0.85 rho=44.00 mass=3452.35
rad_shunt   area=5.4627 t=400.00 tspace=255.00 e=0.85 ho=44.00 mass=240.36
rad_bc      area=68.5710 t=400.00 tspace=255.00 e=0.85 rho=44.00 mass=3017.13
bus_l       power=4.000e+04 voltage=9.996e+01 current=4.002e+02

```

output of model masses

model	mass kg
reac_l	1.385e+03
reac_l.ss	1.000e+02
reac_l.rs	1.466e+03
reac_l.boom	9.000e+01
ti_l	7.363e+02
rad_prim	3.452e+03
rad_shunt	2.404e+02
rad_bc	3.017e+03
mass_sys.dist	2.200e+02
mass_sys.ic	2.220e+02
mass_sys	1.093e+04

APPENDIX G

Performance Map Layouts

Several of the dynamic models make use of performance maps which are read from a file. Each of these performance maps consists of functions of either one or two independent variables. This section describes the layout within the files of this data and the interpolation classes that are used to obtain the data and to perform the interpolations. These interpolations, at present, are only multilinear.

We start with the one dimensional interpolations. This class is used to define z as a function of x given x_i, z_i pairs, where $i=0$ to n , as follows.

$$t = \min(\max(x, x_0), x_n)$$

$$z = z_i + (t - x_i) (z_{i+1} - z_i) / (x_{i+1} - x_i)$$

where i is the largest i such that $x > x_i$, $i < n$. The class structure is defined as

```
struct intp1
{
    char label[16];
    double *x, *z, y;
    int n, im;
};
void intp1new();
void intp1in();
double intp1c();
```

The variables have the following meaning.

- label - user defined name for the data.
- x - pointer to the array of x_i grid values.
- z - pointer to the array of z_i grid values.
- y - additional value for this data; used only for two-dimensional interpolations.
- n - number of elements in the x (and z) arrays.
- im - additional parameter used only internally in the two-dimensional interpolations.

One member function `in` is defined for the class, which is called with a pointer argument to the data structure and with a file descriptor argument to read the data. The data in this file is in the following order separated by blanks or new lines. First, the user defined label is specified, followed by the number n of elements in the x array and then followed by the additional y value. If only a one dimensional interpolation is being done, this y value may simply be set to zero. Following these three items, are the n sets of x, z pairs.

The function `intp1c` for the class defines the function for doing the interpolation. This function takes as arguments a pointer to the data structure and a double precision variable representing the independent variable value. As an example of its use, suppose the file `IN1` contains the data (laid out as specified above) for the pump head as a function of pump rpm. In this case the rpm values would be the x array and the head values the z array. The coding necessary to read in the data and then interpolate to find the head for some rpm, would be as follows.

```
#include "util.h"

main()
{
    double h, rpm=600;
```

```

FILE *inp;
struct intp1 head;
inp=fopen("IN1","r");
intplin(&head,inp);
h=intp1c(&head,rpm);
}

```

Note, the interpolations classes are defined within the "util.h" header file. Here a file descriptor, `inp` is declared and associated with the file named "IN1" using the `fopen` function, then the `head` variable is declared as a `intp1` class instance, the `intplin` function is called with the file descriptor as an argument to read in the head data, then `intp1c` is called to return the value of the head for some rpm value, in this case, 600 rpm.

The reason the file descriptor is used within the `intplin` function rather than the file name is so a single file can be opened and used with several interpolation classes. This is done in the pump model where both the head and the torque curves are defined as `intp1` classes with both sets of data in the same file. The independent argument, however, is not the rpm as in this simple example. More on that later.

The two dimensional interpolations, z as a function of x and y , are performed by interpolating over two fixed y grid values as one dimensional interpolations over x and then interpolating these results as a one dimensional interpolation over y . As an option a curve in the x,y plane can be input that describes some prominent feature of the surface z , say a ridge or valley, and the x interpolations are then performed relative to this feature curve. That is, if the feature curve is denoted, $x=x_{feat}(y)$, then the x interpolations at the two y grid values are made at $x_{feat}(y_{grid})+x-x_{feat}(y)$, where y_{grid} is the y grid values. These two interpolation values are then interpolated as a one dimensional interpolation over y as before.

The two dimensional interpolation class structure is defined as follows.

```

struct intp2
{
    int n, ifeat;
    intp1 *a;
};
void intp2in();
double intp2c();

```

The variables have the following meaning.

- `n` - number of y grid values.
- `ifeat` - flag indicating that the interpolations are to be performed along some feature curve rather than on a rectangular x,y grid.
- `a` - pointer to an array of one dimensional interpolation classes used to hold all of the data. For each of these classes the y variable holds the fixed y grid value.

The use of this two dimensional interpolation class is exactly like the one dimensional class except that the class is called as a function of two variables. The `intp2in` function makes use of the `intp1in` function to obtain the data. The complete layout of the `intp2` data consists of eight dummy numbers (these numbers were originally used to define a graphical representation of the data but are currently not used) followed by the number, n , of y grid values. These are then followed by n sets of one dimensional interpolation data laid out exactly as with the `intp1` class data. The last of these `intp1` sets may have the label, "feature", indicating that this set represents the feature curve. In this case the x values of the data represent the y values of the feature curve, and the z values represent the x values of the feature curve. Other than the use of the "feature" label, the label used on the `intp1` data sets really isn't used for the `intp2` class, although, some dummy label needs to be supplied.

With these interpolation classes defined, we can now consider the actual performance maps that are used by the dynamic pump, compressor, and gas turbine models.

The pump model requires two maps, one for a nondimensional head and one for a nondimensional torque, where a nondimensional quantity is defined by division by its rated value. These nondimensional head and torque curves are defined by

$$H_n = (m_n^2 + rpm_n^2) H(x)$$

$$T_n = (m_n^2 + rpm_n^2) T(x)$$

where

$$x = \pi + \tan^{-1} \left(\frac{m_n}{rpm_n} \right)$$

and the functions, $H(x)$ and $T(x)$, are the normalized head and torque curves at $m_n^2 + rpm_n^2 = 1$. The pump performance map file contains $H(x)$ as the first `intp1` class data and $T(x)$ as the second `intp1` class data.

The compressor model also has two performance maps, one for the pressure ratio and one for the efficiency. This time the maps are functions of two parameters, the corrected mass and corrected rpm, defined as

$$m_{cor} = \frac{\frac{m\sqrt{t}}{p}}{\frac{m_r\sqrt{t_r}}{p_r}}$$

$$rpm_{cor} = \frac{rpm/\sqrt{t}}{rpm_r/\sqrt{t_r}}$$

where the r subscript refers to rated conditions. The corrected mass parameter is treated as the x variable and the corrected rpm as the y parameter in a two dimensional interpolation. Both of these performance maps are stored within the same file with the pressure ratio map specified first.

The gas turbine model has exactly the same set of performance maps (although the maps themselves are different) as the compressor model. Note, that for both the compressor and the gas turbine, the model will perform some additional scaling of the resulting pressure ratio and efficiency as explained within the section describing the models.